Genetic Algorithm Based Estimation of Non–Functional Properties for GPGPU Programs

Adrian Horga[1] Sudipta Chattopadhyay[2] Petru Eles[1] Zebo Peng[1]

[1]Linköping University [2]Singapore Univ. of Tech. and Design

adrian.horga@liu.se, sudipta_chattopadhyay@sutd.edu.sg, {petru.eles, zebo.peng}@liu.se

Abstract

Non-functional properties, like execution time or memory access information, of programs running on graphics processing unit (GPUs) can raise safety and security concerns. For example, understanding the execution time is critical for embedded and real-time applications. To this end, worst-case execution time (WCET) is an important metric to check the real-time constraints imposed on embedded applications. For complex execution platforms, such as GPUs, analysis of WCET imposes great challenges due to the complex characteristics of GPU architecture as well as GPU program semantics. GPUs also have specific memory access behavior. Observing such memory access behavior may reveal sensitive information (e.g. a secret key). This, in turn, may be exploited to launch a side-channel attack on the underlying program.

In this paper, we propose `GDivAn`, a measurement-based analysis framework for investigating the non-functional aspects of GPU programs, specifically, their execution time and side-channel leakage capacity. `GDivAn` is built upon a novel instantiation of genetic algorithm (GA). Moreover, `GDivAn` improves the effectiveness of GA using symbolic execution, when possible. Our evaluation with several open-source GPU kernels, including GPU kernels from the OpenSSL and MRTC benchmark suite, reveals the effectiveness of `GDivAn` both in terms of finding WCET and side-channel leakage.

1. Introduction

In recent times, general-purpose graphics processing units (GPGPU or simply GPU) have seen increased usage in real-time applications due to their high computational capacity and low power consumption. This includes applications in avionics and automotive, among others. Such applications raise safety and security concerns. It is, therefore, natural to investigate non-functional properties related to the safety and security of real-time GPU applications.

Timing behavior is a non-functional property closely related to safety in real-time applications. In this case we are interested in estimating the worst-case execution time (WCET) of real-time GPU applications. The analysis of WCET is crucial to check whether a given real-time application meets the deadline. WCET captures the maximum execution time for a given program over all its inputs. For a given GPU program, its WCET depends on the program and its features as well as the GPU and its features.

From the security point of view, it is necessary to reason whether GPU applications have side-channels that can be used to reveal secret information (e.g. a secret key). More importantly, it is critical to determine how much information does a given GPU implementation leak to an attacker through a side channel. The more unique observations an attacker can make through the targeted side-channel, the less secure the GPU program is in respect to the sensitive information it contains.

In this paper, we propose `GDivAn`, a measurement-based analysis framework for arbitrary GPU kernels for investigating the non-functional aspects of GPU programs, specifically, their execution time and side-channel vulnerability. In order to distinguish, in the `GDivAn` framework, between the timing analysis and side-channel vulnerability analysis components, we refer to the two as `GDivAn_T` and `GDivAn_C`. In our previous work [1], we proposed `GDivAn_T` to systematically test the WCET of GPU kernels. In this work, we extend the capability of the `GDivAn` framework to not only measure WCET, but also to measure its vulnerability against side-channel attacks with the use of `GDivAn_C`.

WCET analysis

The design and implementation of `GDivAn_T` involves several technical challenges for WCET analysis. Firstly, GPU programs employ massive parallel processing. Hence, it is infeasible to generate all possible execution scenarios in realistic GPU programs. To this end, we first symbolically execute a GPU program with a small number of threads. We employ such a strategy to hypothesize that it is often possible to cover the GPU program features (e.g. all possible branch outcomes) with a small number of threads. The small number of threads will also help mitigate the limitations of symbolic execution such as path explosion [2]. Secondly, GPUs involve complex micro-architectural features, involving shared memory, hundreds of processing units and caches. Moreover, the exact nature of these micro-architectural features (e.g. cache replacement, thread scheduling and bus arbitration policies) remain opaque to developers. This, in turn, makes the works on static WCET analysis [3, 4, 5] practically infeasible for GPGPU programs. To solve such challenges, we directly execute the generated test inputs in commodity GPU-based systems. Thus, the key novelty in our approach is to drive the generation of such test inputs to expose the WCET. To this end, we leverage the results obtained from symbolically executing a GPU program with a small number of threads. Subsequently, we use these results as initial information for a genetic algorithm. The genetic algorithm will then systematically search the input space and potentially converge towards the WCET.

Side-channel analysis

For determining the amount of information leaked through a side-channel, the sensitive information must influence the observed values through the side-channel. If changes in the sensitive information do not lead to changes in the observed side-channel information, then the GPU program could be safe from attacks from the respective side-channel. However, the more unique observations an attacker can make for different values of the sensitive information, the less secure that particular GPU implementation is. GPU programs that contain sensitive information include GPU implementations of cryptographic algorithms, like the Advanced Encryption Standard (AES). Such algorithms typically do not have branching statements to avoid influencing the control flow via secret inputs. Consequently, it is unnecessary to use the heavy machineries underneath a symbolic execution to explore the GPU code. Nevertheless, the possible state space for the values of the sensitive information still remain too large for an exhaustive search. We employ `GDivAn_C`, with its tailored genetic algorithm, to systematically search for sensitive information values that will provide unique observations through the targeted side-channel. The genetic algorithm in `GDivAn_C` has been tailored in such a fashion that it maximizes the potentially observed values through side channels.

This paper presents the following contributions:

1. We propose `GDivAn`, a novel approach that employs a synergistic combination of symbolic execution (SE) and genetic algorithm (GA). This is to analyze the non-functional properties: timing (`GDivAn_T`) and side-channel leakage (`GDivAn_C`) of arbitrary GPU programs (Section 5).
2. We implement `GDivAn` for commodity GPU hardware (i.e. Nvidia Tegra K1 GPU, Nvidia GTX 1060M GPU). Such an implementation can easily be integrated with any measurement-based non-functional analysis for GPU programs.
3. We evaluate `GDivAn` for several GPU kernels involving up to hundreds of thousands of threads (Section 6). Our evaluation reveals that `GDivAn_T` is significantly more effective in exposing the WCET compared to random testing, genetic algorithm in isolation and the state-of-the-art fuzz testing tool Radamsa [6]. Also, `GDivAn_C` can produce more unique side-channel

observations compared to random testing and Radamsa. Our implementation and all experimental data will be publicly available.

2. Related work

How does `GDivAn_T` differ from the state-of-the-art in timing analysis? Existing works in timing analysis of GPU applications [7, 8] address the system-level schedulability analysis assuming the WCET of individual programs as given and ignoring how it can be produced. As opposed to this, we focus on the structure of individual GPU programs and produce their WCETs. In the past few years, there has been a rise on using extreme value theory (EVT) for measurement-based timing analysis [9, 10, 11]. However, there are strong requirements to be satisfied for any result of an EVT-based approach to produce a sound WCET [11]. Such is the collection of a representative input sample that needs to satisfy key assumptions like independence and identical distribution. While techniques have been proposed to achieve some of these requirements, they need deep interventions in the hardware/-software architecture [11] and are not applicable to processors of the complexity typical to a GPU. In general, even in the case of less complex, classical architectures there are several open challenges to be solved before EVT-based techniques can be generally applied and trusted [12, 13, 14]. In general, existing works on the WCET analysis for GPUs [9, 10, 15] ignore the systematic generation of test inputs for exposing the WCET and thus ignore the effects branching and branch divergence might have on the overall WCET of GPU programs. In addition, hybrid WCET analysis [15] is not applicable for commodity GPU-based systems. This is due to its implicit assumption on the availability of the GPU execution model which, as mentioned before, is not the case.

Parametric WCET analysis combined with genetic programming [16] provides an expression of the WCET. However, assumptions regarding the underlying hardware must be made in order to compute the WCET expression. In GPUs, this approach is not feasible at the moment due to lack of precise hardware models.

The work [17] uses branch coverage as leverage for the genetic algorithm for WCET analysis of GPU programs. Obtaining branch coverage is indeed more feasible than obtaining path coverage via symbolic execution. Nevertheless, such an approach captures only a subset of the paths captured by symbolic execution. Thus, the risk of missing critical paths in the program is increased by the usage of branch coverage.

It is worthwhile to mention, that the genetic algorithm of `GDivAn_T` could be complemented by using meta level parameter tuning [18] to determine the best possible parameters for WCET analysis of the targeted GPU programs. This would, however, increase the execution time of such analysis by several orders of magnitude.

There has been effort [19] from the software engineering community to combine symbolic execution and genetic algorithm. However, such effort is focused on the functionality testing of software. In contrast, GDivAn focuses on testing non-functional properties. Specifically, GDivAn_T sets itself apart from existing works on combining symbolic execution and genetic algorithm [20, 19], by its novel mechanism for testing the WCET of GPU programs.

How does GDivAn_C differ from the state-of-the-art in side-channel analysis? GPU implementations of cryptographic algorithms have been shown to be vulnerable to side-channel attacks for retrieving the secret key. The side-channels used range from shared memory [21] and cache [22] to power profiles [23]. However, these works target a specific GPU implementation and do not consider the vulnerabilities of different implementations for the same side-channel. Our work aims to complement such approaches by providing an estimation tool that can quantify the leakage of a side-channel (i.e. shared memory) and rank different GPU implementations of the same algorithm. This can subsequently help users in choosing the least vulnerable implementation.

The work [24] leverages the vulnerabilities of CUDA [25] primitives for allocating GPU memory in order to read secret information directly from GPU memory. Targeted memory locations ranged from shared memory to global memory and registers. Some of the presented vulnerabilities could not be found in newer CUDA enabled GPUs. Our work is orthogonal to such approaches. In contrast to exploiting vulnerabilities in CUDA primitives, we aim to devise a test generation methodology that quantifies the side-channel vulnerability of GPGPU programs. To this end, we focus on shared memory bank conflicts. However, our GDivAn_T approach can easily be generalized for other side channels.

Works for quantifying side-channel leakage have been done for CPU implementations [26, 27, 28, 29, 30], targeting cache-related side-channels. In contrast to these works, we focus on the shared memory side-channel. Shared memory is a type of memory available to most GPU platforms. Such a memory is controlled by the programmer, as opposed to the cache. Moreover, as observed in existing works [21], the number of shared memory transactions may reveal secret information from cryptographic applications. Thus, to understand the vulnerability against shared memory side channel attacks, it is crucial to understand the dependency between a secret input and the number of shared memory transactions. This requires different mechanisms as compared to the analysis of cache side-channel behavior.

In summary, to the best of our knowledge, this is the first work aimed at quantifying shared memory side-channel leakage on GPU platform.

3. System and Execution Model

In this paper, we target GPU kernels written in CUDA [25] [1] and execution platforms similar to NVIDIA GPUs. However, we believe that the core capabilities within `GDivAn` are also applicable to other GPU platforms. The smallest execution unit in CUDA programs is a *thread* and the program running on the GPU is called a *kernel*. Several threads can be grouped into a *thread block*. GPUs use Streaming Multiprocessors (SMs) to execute the kernels assigned to them. Each SM has its own memory subsystem. Such a memory subsystem typically involves registers, scratchpad memories and multiple levels of caches. All the SMs have access to the global memory and all the threads within a thread block run on the same SM. SMs leverage the single-instruction-multiple-threads (SIMT) paradigm to employ large-scale parallelism. To this end, threads in a thread block are grouped into *warps*. All the threads within a warp execute instructions in lock-step.

Typically SMs in a GPU do not employ branch prediction. If two threads of the *same warp* activate different targets of a branch instruction, then a phenomenon, commonly known as *branch divergence*, takes place. Branch divergence may significantly affect the level of parallelism offered by GPUs. For a typical "`if (C) then A else B`" structure, branch divergence leads to a serial executions of `A` and `B`. Threads that do not activate the *true* leg of conditional `C` are disabled while `A` is executed. Likewise, all threads satisfying the conditional `C` are disabled while `B` is executed.

Shared memory behavior

In CUDA programming, scratchpad memory is referred to as shared memory. All the threads in a thread block have access to the same shared memory. In contrast to caches, shared memories are managed by the programmer. The memory subsystem has several banks that manage parts of the shared memory. As described before, threads are grouped into warps. A warp can load all the requested data from shared memory in one transaction if each request is serviced by a different bank. However, if the requests need to be served by the same memory bank, then the requests to the bank are serialized. Therefore, the number of transactions for a given access to the shared memory is equal to the maximum different requests steered to a given memory bank. The number of shared memory transactions can be monitored (e.g. via a profiler) and resembles a side channel [21] that can be exploited to ex-filtrate sensitive information (e.g. a secret key).

[1] `GDivAn` is equally applicable to other GPU programming paradigms like OpenCL

4. Overview

In this section, we discuss the challenges in estimating WCET of GPU-based programs via simple examples. Subsequently, we show the key insight behind our approach for `GDivAn_T` for WCET analysis. We also discuss how observing memory-access information for a GPU program can be used in a side-channel attack and how `GDivAn_C` can quantify the vulnerability from such memory-access information.

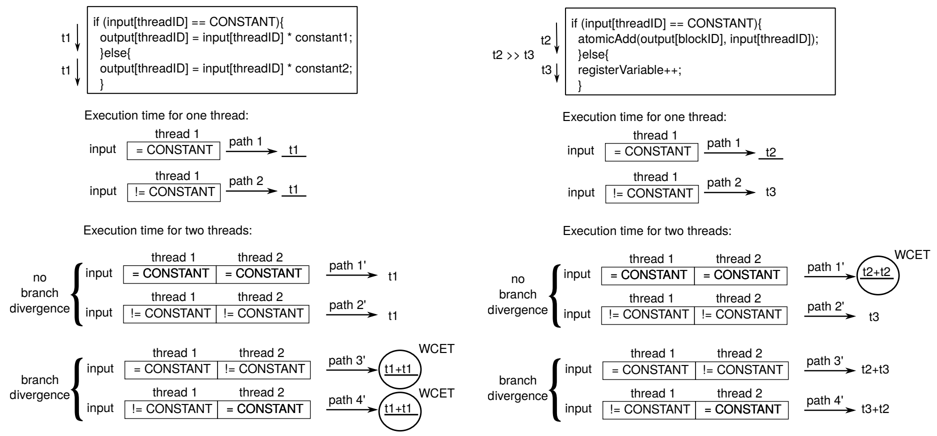
4.1. Challenges in WCET analysis for GPU-based programs

Consider the GPU kernels shown in Figure 1. For the sake of simplicity in this example, we assume that any pair of threads, executing the same instruction in the kernels, are free from memory contention. In Figure 1(a), let us assume that the multiplication instruction takes time $t1$ to execute. Hence, for each thread, the WCET is $t1$. However, GPU kernels typically involve thousands of threads running in parallel. If the kernel in Figure 1(a) is executed for two threads, then it leads to four different execution scenarios depending on the value of `input`. As shown in Figure 1(a), the WCET (*i.e.* $t1 + t1$) is manifested for $path\ 3'$ and $path\ 4'$. This is due to the divergence that takes place at the branch instruction.

From the discussion in the preceding paragraph, we may hypothesize that branch divergence always leads to longer execution time. The example in Figure 1(b) however, contradicts this hypothesis. In Figure 1(b), the true leg of the branch involves an atomic add operation, which, in turn might access the slow global-memory. In contrast, the false leg of the branch involves simple arithmetic manipulations on registers. Accessing global-memory is several orders of magnitudes slower than accessing register variables. Hence, in Figure 1(b), $t2 \gg t3$, where $t2$ ($t3$) is the time to execute the true (false) leg of the branch. If two threads execute the kernel in Figure 1(b), then the WCET is $t2 + t2$, due to the atomic nature of the operation which imposes serialization. We note that the WCET is manifested for an execution scenario that does not exhibit branch divergence. Intuitively, this occurs due to the unbalanced execution time across different legs of the branch instruction.

Will random testing work? In Figure 1, the number of execution scenarios grows exponentially with the number of threads. In particular, consider to use random testing for exposing WCETs of the examples in Figure 1. We observe that random testing only has a slim chance (probability $< 0.4\%$ [2] for two threads) to synthesize

[2] If the element is stored on the minimum of one byte (8 bits), the probability percentage that its value will be equal to the $CONSTANT$ is $\frac{1}{2^8} \cdot 100 = \frac{1}{256} \cdot 100 < 0.4\%$



```
if (input[threadID] == CONSTANT){
  output[threadID] = input[threadID] * constant1;
  }else{
  output[threadID] = input[threadID] * constant2;
  }
```

```
if (input[threadID] == CONSTANT){
  atomicAdd(output[blockID], input[threadID]);
  }else{
  registerVariable++;
  }
```

t1
t1
Execution time for one thread:
thread 1
input = CONSTANT path 1 t1
input != CONSTANT path 2 t1
Execution time for two threads:
thread 1 thread 2
no branch divergence
input = CONSTANT = CONSTANT path 1' t1
input != CONSTANT != CONSTANT path 2' t1
branch divergence
input = CONSTANT != CONSTANT path 3' t1+t1 WCET
input != CONSTANT = CONSTANT path 4' t1+t1 WCET
t2
t2 >> t3
t3
input = CONSTANT path 1 t2
input != CONSTANT path 2 t3
input = CONSTANT = CONSTANT path 1' t2+t2 WCET
input != CONSTANT != CONSTANT path 2' t3
input = CONSTANT != CONSTANT path 3' t2+t3
input != CONSTANT = CONSTANT path 4' t3+t2

(a) Balanced branch

(b) Unbalanced branch

Figure 1: **Motivational example**. $thread$ 1 and $thread$ 2 belong to the same warp. Both examples use one input and one output vector. Both vectors are stored in GPU global-memory. The variable $threadID$ captures the global identity of a thread within the execution. For each thread, variable $blockID$ captures the identity of the thread block. Threads in a warp belong to the same thread block.

input vectors equal to $CONSTANT$. Since the WCET is manifested only for such input vectors, it is unlikely that random testing will converge towards exposing the WCET of programs in Figure 1.

Will symbolic execution work? Symbolic execution poses an attractive choice to systematically explore all unique execution paths in an application. It leverages the power of constraint solvers to symbolically capture all inputs exhibiting an execution path. Then, such a symbolic formula is manipulated to generate inputs for a different execution path. As a result, if our example programs in Figure 1 are executed with two threads, symbolic execution terminates generating four test inputs – one each for a unique execution scenario. Thus, the probability to expose WCET in our example program increases to 100% if all four symbolically detected paths are executed.

Unfortunately, the complexity of symbolic execution may quickly become intractable with the growing number of threads. As GPUs are targeted to support massive multiprocessing, typically GPGPU programs involve thousands of threads. As a result, it is infeasible to explore all possible execution paths (via symbolic execution) for any realistic GPGPU applications.

Key insight. Our GDivAn_T approach proposes a novel mechanism to circumvent the inherent complexity of symbolic execution, yet uses its power to cover the structure of GPGPU programs. Our key intuition is to run a GPU kernel symbolically only for a limited number of threads and to cover all (or most) execution paths of this kernel. Subsequently, we investigate each path with respect to some program features (e.g. number of instructions) that influence timing. Leveraging an SMT solver, we generate test inputs (atoms) for each path. Finally, we systematically scale and manipulate these test inputs for the original kernel that potentially runs with significantly larger number of threads. The final stage involves a novel application of genetic algorithm to manipulate the test inputs. In essence, our GDivAn_T approach combines the strength of symbolic execution to explore program structure and the strength of genetic algorithm to systematically search a large input space.

How GDivAn_T works. Figure 2 provides an outline of GDivAn_T. At a high level, the workflow of GDivAn_T involves three steps: 1) generation of input atoms, 2) scaling of atoms, and 3) exploration of input space via genetic algorithm.

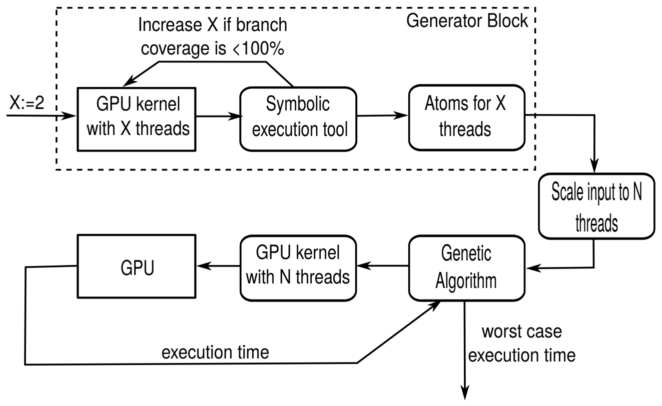


Figure 2: Overview of GDivAn_T

1) Generation of input atoms: Consider the GPU kernel shown in Figure 1(a). We use the predicate $pred_e(th)$ to symbolically capture the execution of control flow edge e in thread th. Concretely, $pred_e(th)$ is *true* if control flow edge e is executed in thread th. Otherwise, $pred_e(th)$ is set to *false*. Let us assume that the *true* and *false* legs of the conditional in Figure 1(a) capture control flow edges $e1$ and $e2$, respectively. In order to generate input atoms, we symbolically execute a given GPU kernel with a small number of threads. For instance, using two threads (say, thread 0 and thread 1), a symbolic execution of the code in Figure 1(a) will result in the following set of execution paths: *(a)* $path_1 : pred_{e1}(0) \wedge pred_{e1}(1)$, *(b)* $path_2 : pred_{e2}(0) \wedge pred_{e2}(1)$, *(c)* $path_3 : pred_{e1}(0) \wedge pred_{e2}(1)$, and *(d)* $path_4 : pred_{e2}(0) \wedge pred_{e1}(1)$. We note that $path_{1...4}$ symbolically captures all inputs leading to the respective execution paths. We leverage on constraint solvers to generate input atoms from these symbolic formulas. The primary goal of this step is to explore the GPU-kernel structure for a small number of threads (i.e. two threads in this example) and generate representative test inputs covering the structure of the GPU-kernel. Our intuition is that it is often feasible to cover the structure of GPU kernels (e.g. all branches in the kernel) despite being executed with a small number of threads.

2) Scaling input atoms: Input atoms, as discussed in the preceding paragraphs, can be used to execute the given GPU kernel with a small number of threads. To obtain an initial set of test inputs for the original kernel, which typically executes with thousands of threads, we systematically scale the input atoms obtained via symbolic execution. Figure 3 outlines input atoms generated from symbolically executing the code (also shown in Figure 3) with two threads. Subsequently, these input atoms were scaled, as shown on the top right corner of Figure 3.

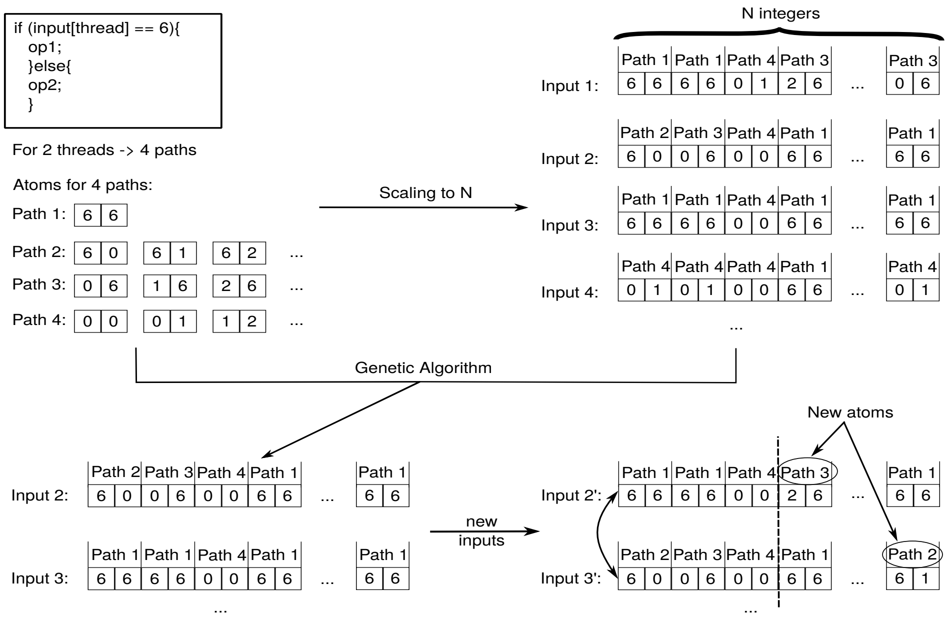


Figure 3: Examples on scaling to large inputs and creating new inputs during the Genetic Algorithm stage in `GDivAn_T`

3) Exploration of input space via genetic algorithm: Steps 1) and 2) generate test inputs that reduce the search space. However, the paths that heavily affect the execution times through branch divergence, instruction serialization or cache contention need to be detected from the large remaining search space. We leverage genetic algorithm (GA) to systematically explore this search space. To this end, we first run the given GPU kernel with the set of scaled inputs (obtained from the previous stage) and obtain the execution time for each such input. Based on the execution times obtained, GA builds new inputs via a series of selection, crossover and mutation operation on the current set of test population and input atoms obtained via the symbolic execution. Our objective in the GA is to maximize the execution time of the GPU kernel. Figure 3 contains an example of two new input vectors. These input vectors were created using two of the current input vectors and input atoms, as shown in Figure 3. The process of executing test inputs and generating new test population via GA is repeated until the execution time does not show significant variation across two consecutive generations of GA.

4.2. Quantifying side-channel leakage for GPU programs

Information about cache accesses [22] and shared-memory accesses [21] can be used as side channels for recovering critical data such as secret keys from GPU implementations of encryption algorithms. Concretely, an attacker monitors the number of cache misses or the number of shared memory transactions and employs statistical techniques to discover secret information. Intuitively, if the number of cache misses or shared memory accesses depend on the secret information, then such information may leak through the respective side channels.

To formalize the notion of side channel explained in the preceding paragraph, we assume that a side-channel to be a function $C : \mathbb{I} \rightarrow \mathbb{O}$. The function C maps a finite set of sensitive inputs to a finite set of observations made by the attacker. If the attacker monitors, for example, shared memory transactions, an observation $o \in \mathbb{O}$ captures the number of shared memory transactions in an execution. If we model the choice of a secret input via a random variable X and the respective observation by a random variable Y, the leakage through channel C is the reduction in uncertainty about X when Y is observed. The maximal leakage through channel C can be defined as follows [26]:

$$ML(C) \leq log_2|C(\mathbb{I})| \tag{1}$$

where $ML(C)$ captures the maximal leakage of channel C. In Equation 1, equality holds when X is uniformly distributed. Since we aim for a software validation framework, we assume the presence of a strong attacker whose choice of secret input is uniformly distributed. Therefore, $ML(C)$ is maximized and $ML(C) \leq$

$log_2|C(\mathbb{I})|$ holds (cf. Equation 1). As a result, the number of unique observations by the attacker (i.e. $|C(\mathbb{I})|$) resembles the side-channel leakage of the respective program.

From the viewpoint of testing, `GDivAn_C` acts as a coverage criterion. Concretely, the higher the value of $|C(\mathbb{I})|$, the better is the effectiveness of `GDivAn_C`. Thus, we aim for `GDivAn_C` to maximize the value of $|C(\mathbb{I})|$. This means that we generate test inputs in order to explore as many unique observations as an attacker can make. For this purpose, we leverage the state space exploration capabilities of the genetic algorithm, modified for exploring and maximizing the unique observations.

Figure 4 outlines the components of `GDivAn_C`. The genetic algorithm provides inputs for the GPU kernel. The kernel is executed on the GPU. During the kernel execution, the GPU profiler captures the value of the targeted metric an attacker might use for a side-channel attack. The observed value is, then, provided back to the genetic algorithm, which uses the past observation and the current one, to generate new inputs that will maximize the number of unique observations. At the end of the execution for `GDivAn_C`, the total number of unique observed values measured for the side-channel are reported as a quantity of the selected side-channel leakage vulnerability.

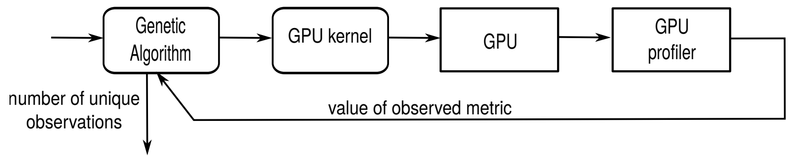


Figure 4: Overview of `GDivAn_C`

In the next section we present the usage of the `GDivAn` framework for WCET analysis (`GDivAn_T`) and side-channel leakage analysis (`GDivAn_C`).

5. Detailed Methodologies

5.1. *WCET analysis using `GDivAn`*

In this section, we describe the mechanism of different building blocks in `GDivAn` for WCET analysis (`GDivAn_T`). The overall outline of `GDivAn_T` and the inter-dependencies between its building blocks are given in Figure 2.

5.1.1. *Generator block*

The purpose of this block is to enable the creation of an initial test population. To this end, we generate test inputs via symbolic execution.

The test inputs are generated to cover the structure (e.g. the branches) of a GPU kernel. As discussed in Section 4, it is practically infeasible to employ symbolic execution for realistic GPU kernels running a large number of threads. Hence, we apply symbolic execution in a trimmed down version of a given GPU kernel. Such a trimming is employed by symbolically executing the GPU kernel only with a small number of threads and aiming to obtain *branch coverage*. We describe this in the following.

Trimmed symbolic execution

We note that it requires at least two threads to manifest branch divergence in a GPU kernel (see Figure 1). In Figure 1(a), we observed that the presence of branch divergence may lead to longer execution time. The primary intuition behind obtaining the branch coverage is to have representative inputs in the test population that trigger branch divergence. To employ our trimmed version of symbolic execution, we systematically increase the number of threads (starting from two threads) in the GPU kernel and invoke the symbolic execution engine. For each invocation of the symbolic engine, we measure the branch coverage being obtained. Finally, we stop the symbolic execution process once both legs of all branch instructions are covered. However, for complex GPU kernels, it might even be infeasible to obtain 100% branch coverage within a reasonable time. For such cases, we impose a time bound (<1 hour) on the symbolic runs of the kernel.

Let us assume that symbolic execution of the kernel is performed for X number of GPU threads. Upon termination of the symbolic execution, it generates the set of all execution paths in the kernel trimmed down to X threads. For each explored path π, we collect the following information:

$$P_{\pi}(X) \equiv \langle form_{\pi}, brdiv_{\pi}, instr_{\pi} \rangle \tag{2}$$

Where $form_{\pi}$ symbolically captures all inputs leading to the execution path π, $brdiv_{\pi}$ captures the total branch divergence along the path π and $instr_{\pi}$ is the total number of instructions executed along π.

We compute $brdiv_{\pi}$ per barrier interval. A barrier interval is the code between the start of the kernel and the first barrier instruction or between two consecutive barrier instructions. The end of a kernel serves as an implicit barrier.

For each execution path π explored via symbolic execution, the branch divergence per barrier interval b is computed as follows:

$$brdiv_{\pi}(b) = \frac{divergent\_sets_b}{X-1} \times 100$$

Where $divergent\_sets_b$ is the number of different paths taken by the X threads in the given barrier interval b. Finally, the total branch divergence is computed by summing up the branch divergence over all barrier intervals ($|BI|$ captures the number of barrier intervals):

$$brdiv_{\pi} = \frac{\sum_{b=1}^{|BI|} brdiv_{\pi}(b)}{|BI|}$$

Branch divergence and the number of instructions per execution path serve as important information for guiding the test generation. In general, the genetic algorithm systematically leverages the information on branch divergence and number of instructions in order to generate test inputs maximizing kernel execution time.

5.1.2. Scaling

In this stage, we scale the input atoms generated via symbolic execution to fit the input size of the given GPU kernel. Assuming that the GPU kernel involves N GPU threads, recall that we run the symbolic execution for $X \leq N$ threads. As a by-product of symbolic execution, we obtain a set of paths (captured by symbolic formulas) in the GPU kernel involving X number of GPU threads. The key intuition of this scaling process is that often the different paths in the given GPU kernel (that involves N threads) can be generated via the combinations of paths obtained from its trimmed version (that involves X threads). For instance, consider our examples in Figure 1. Assume $path_n^i$ captures the i-th path in the kernel involving n threads. The following relationships hold:

- $path_2^1 = path_1^1 \parallel path_1^1$
- $path_2^2 = path_1^2 \parallel path_1^2$
- $path_2^3 = path_1^1 \parallel path_1^2$
- $path_2^4 = path_1^2 \parallel path_1^1$

where $\parallel$ captures an ordered (with respect to thread identities) combination of different execution paths. Ideally N is divisible by X so that no truncation is needed while scaling the input atoms.

It is worthwhile to mention that branch conditionals targeting thread ID (i.e. $(\mathit{threadID} == \mathit{CONSTANT})$) or positioning of the thread (i.e. $(\mathit{threadID} == \mathit{inputLength})$) are special cases that cannot be handled via the scaling process. For instance, when scaling from X threads to $2 \cdot X$ threads, thread X will no longer be equal to the input length. Hence, our trimmed version of symbolic execution will not cover all branch legs for such conditional branches involving input lengths or specific thread IDs. Hence, we complement both the symbolic execution and the scaling process via a genetic algorithm. This is to systematically explore the input space for covering worst-case scenarios that were not obtained after the scaling.

5.1.3. Genetic Algorithm

Our engineered algorithm involves a mapping between terms used in the genetic algorithm [31] (GA) literature and the specific context of our targeted problem. Table 1 provides an outline of this mapping and we use the terms used in Table 1 for the rest of the discussion. Our GA process is outlined in Figure 5.

WCET problem	GA term
Path formula (i.e. $form_\pi$)	Allele
Part of kernel input	Gene
Kernel input	Individual/Chromosome
Kernel execution time	Fitness

Table 1: Mapping the WCET problem to genetic algorithm terms

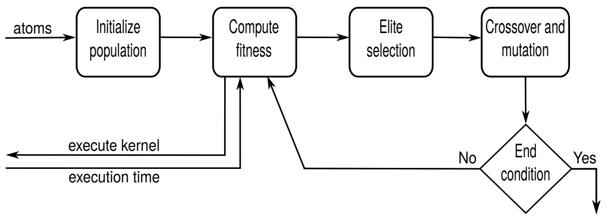


Figure 5: Stages of the Genetic Algorithm for `GDivAn_T`

An individual is mapped to an input for the kernel. If the kernel runs for N threads and the trimmed symbolic execution runs for X threads, then we encode the chromosome as a combination of $\frac{N}{X}$ genes. Figure 3 captures this phenomenon. In particular, instead of N genes, we have $\frac{N}{2}$ genes to construct $Input\ 1$ as follows:

$$Input\ 1 = Path\ 1 \| Path\ 1 \| Path\ 4 \| Path\ 3 \| \ldots \| Path\ 3$$

To maintain diversity in the initial test population, we employ multiple strategies. Firstly, we create individuals by combining a set of randomly selected paths. These paths belong to the trimmed version of the kernel. Secondly, we select individuals by combining paths that manifested maximum number of instructions and branch divergence in the trimmed kernel. Recall that the information on the number of executed instructions and branch divergence was collected during symbolic execution (*cf.* Equation 2). As an example, assume that the path "$Path\ 2$" exhibits the maximum number of instructions and branch divergence. Therefore, we create an individual as follows: $Path\ 2 \| Path\ 2 \| \ldots \| Path\ 2$. The individuals in the initial population, that are not suitable to expose the WCET, are removed in subsequent iterations via the natural selection of the genetic algorithm.

The elite selection stage selects a predefined percentage of individuals from the population based on their fitness (i.e. the kernel execution time). These individuals are kept unaltered to use in the next population.

During crossover, we select one of the parents randomly and the other with a bias towards the elite. Subsequently, an 1-point crossover is employed between parents. An example of such an 1-point crossover can be observed in Figure 3 (in the bottom half). A small fraction of individuals are mutated at every iteration of the genetic algorithm. To this end, we implement a low-cost mutation operation. Concretely, we generate two random numbers – the first number to check if we hit the probability to mutate and the other (in case we mutate) to identify the specific gene to mutate.

The iterative process of GA continues until the kernel execution times, manifested by two consecutive generations of GA, do not change substantially. The process is also terminated if the time budget of testing is reached.

5.1.4. Why `GDivAn` works for WCET analysis?

The reason `GDivAn` works is because of the synergistic combination of symbolic execution (SE) and genetic algorithm (GA) employed in `GDivAn_T`. The purpose of our symbolic execution step is not to explore all different aspects in the GPGPU program that may impact the program performance. Due to the complexity of symbolic execution, such a strategy is unlikely to scale. Besides, symbolic executors are classically not designed to explore the performance behavior of GPGPU programs. As a result, integrating GPU-specific performance features into a symbolic executor would require heavy engineering of state-of-the-art symbolic execution tools. To address such challenges, we propose to use off-the-shelf symbolic executors and explore the structure of GPGPU programs considering branch divergence. This leads to an initial population of test cases for our genetic algorithm. We are testing each generated test input on real hardware during the genetic algorithm stage. We, therefore, do not solely rely on the assumptions gathered from the trimmed symbolic execution stage that a single path atom could be scaled and provide the test input that produces the WCET. The role of our genetic algorithm is to, then, search the input space and discover inputs that lead to longer execution times due to other micro-architectural features (e.g. thread scheduling, memory coalescing, memory-bank conflicts and cache misses). This makes `GDivAn` a scalable and effective framework to discover the likely WCET of arbitrary GPGPU programs.

`GDivAn` *limitations for WCET analysis*

The complexity of GPU programs especially due to the large number of threads imposes some limitations on our `GDivAn_T` approach. It is a known fact that symbolic execution suffers from path explosion and path divergence [2]. We mitigate such limitations of symbolic execution by using the trimmed symbolic execution method (see Section 5.1.1) to generate input atoms. As mentioned in Section 5.1.2, if branch outcomes depend on the thread ID or the thread positioning, the trimmed symbolic execution embodied in `GDivAn_T` might not reach the respective branch legs. If, during the trimmed symbolic execution stage we did not achieve 100% branch coverage, the genetic algorithm stage could help reach the branches not covered. However, we cannot provide such guarantees. Finally, `GDivAn_T` uses a measurement based approach, therefore, we cannot guarantee the tightness of the obtained WCET. Specifically, `GDivAn_T` does not provide tightness guarantees on the accuracy of the measured WCET, as there is no existing solution that can safely guarantee such tightness.

5.2. Side-channel analysis using `GDivAn`

As discussed in Section 4.2, quantifying the side-channel leakage depends directly on how many different observations a strong attacker can make for the respective side-channel. Cryptographic algorithms implemented on GPUs like AES are appropriate targets [22, 21] for attackers. Such cryptographic algorithms do not usually have key-dependent branching statements. This is to avoid execution behavior dependent on the key. For example, different paths, depending on the key, may lead to vastly dissimilar timing behaviors. This, in turn, may significantly narrow down the search space for an attacker aiming to find the key.

The lack of branching statements in cryptographic algorithms significantly reduces the effectiveness of the symbolic execution part of `GDivAn` employed for WCET analysis (`i.e. GDivAn_T`). However, given a proper fitness function, the genetic algorithm can be used to systematically explore the search space in these single-path algorithms. Concretely, the goal of the genetic algorithm is to search for inputs that maximize the number of observations for the targeted side-channel. For the purpose of this paper, we use the distinct counts of shared memory transactions as the side-channel. This is because such side channels in GPUs are exploited in existing works [22, 21]. Table 2 outlines the mapping of the side-channel problem onto the genetic algorithm terms.

For GPU implementations of table based cryptographic algorithms, such as AES, GPUs can store search tables in the shared memory. This significantly reduces the execution time of the implementation, as the shared memory is two orders of magnitude faster than global GPU memory (DRAM). However, the accesses to

Side-channel problem	GA term
Input byte domain	Allele
Input byte	Gene
GPU program input (secret key)	Individual/Chromosome
Targeted GPU metric (number of shared memory transactions)	Fitness

Table 2: Mapping the GPU side-channel testing problem to genetic algorithm terms

the shared memory are performed based on the value of the shared key. This, in turn, makes a program's shared memory transactions dependent on the value of the secret key. For the genetic algorithm embodied within `GDivAn_C`, each individual corresponds to a different secret key. The final goal of the algorithm is to find as many different secret keys as possible (within a time budget) such that the set of observations (i.e. shared memory transactions) is maximized. In order to accomplish this, the genetic algorithm from `GDivAn_C` needs to be tailored for maximizing the cardinality of the fitness observations set. In other words, `GDivAn_C` needs to search for new and different fitness values. This is in contrast to the exploration strategy for the WCET analysis where `GDivAn_T` needed to search for new but only higher timing values.

The proposed strategy is exploiting information regarding properties of secret keys already explored. This allows us to avoid exploring already available observations. Concurrently, the memory of the past secret keys allows selecting appropriate individuals for future populations. Figure 6 illustrates the steps of the modified GA for the purpose of maximizing the observations set. Obs holds the set of all observations explored by the GA at any point of time. Thus, the cardinality of Obs captures the side-channel leakage of the GPU program under test. In the subsequent sections, we discuss the different stages of this genetic algorithm in more detail.

Population initialization

In the population initialization stage, we randomly generate a set of N individuals (Ind_i, $i \in [1..N]$) and we compute their fitness f_i. We attach to each individual an effective rank er_i calculated as follows:

$$er_i = \begin{cases} 0, f_i \in Obs \\ 1, f_i \notin Obs \end{cases}$$

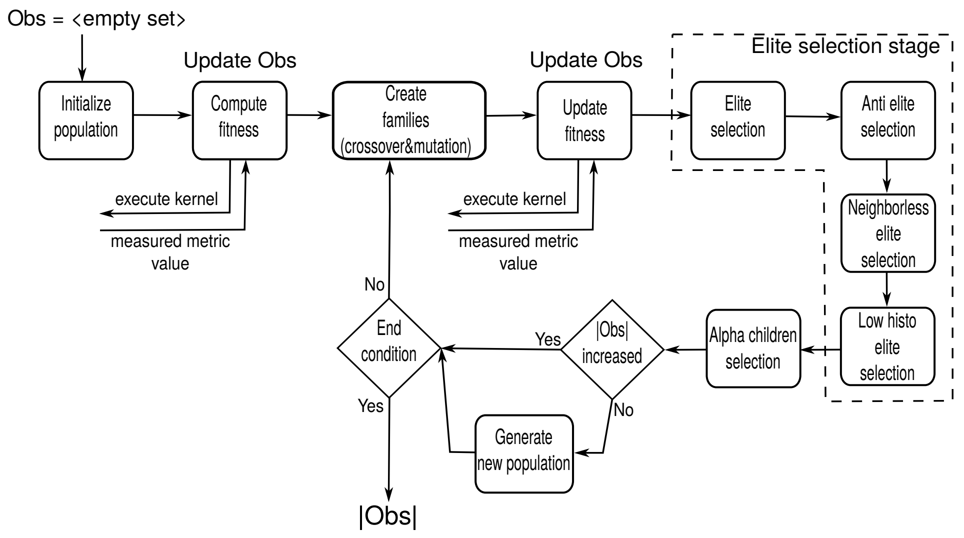


Figure 6: Stages of the modified Genetic Algorithm for `GDivAn_C`

In the initialization step, the effective rank is equal to one for all the individuals, as the Obs set is empty. The effective rank is used as part of a ranking score in the elite selection stage. Moreover, the rank keeps track of the chosen individuals. We note that the effective rank of an individual drops to zero in subsequent iterations if the observations resulting from the individual is already in Obs.

Crossover and mutation (family generation in GA)

In the crossover and mutation stage for `GDivAn_C`, we generate children for each individual. We refer to the list of children generated from the crossover and mutation of Ind_i with other individuals as the family Fam_i of Ind_i. Let us assume $Fam_i = \{Ch_{i1}, Ch_{i2}, \ldots, Ch_{ik}\}$. We compute the fitness (i.e. number of shared memory transactions) of each child Ch_{ij} as f_{ij}. This is performed by running the program with input Ch_{ij} and obtaining the number of shared memory transactions. It is worthwhile to mention that we can also generate children by mutation of the parent chromosomes. The intuition is that a small change in the parent chromosome will be directly reflected in a small change in the observed fitness value. Thus, such mutations are useful for searching values in the neighborhood of the fitness value for the parents. In contrast to the GA from the WCET analysis, the children plays a role in selecting the elites for the next iteration.

For a family Fam_i, we compute two rankings: the family ranking fr_i and the set ranking sr_i of individual Ind_i. The family ranking fr_i represents the cardinality of the set of fitnesses for all the children of Ind_i and is computed as follows:

$$fr_i = \left| \bigcup_{j=1}^{k} \{ f_{ij} \mid Ch_{ij} \in Fam_i \} \right|$$

fr_i is useful to see how diverse individuals a specific family can produce. The set ranking sr_i represents the cardinality of the set of fitnesses for all the children of Ind_i whose fitness is not already in the observed set Obs; sr_i is computed as follows:

$$sr_i = \left| \bigcup_{j=1}^{k} \{ f_{ij} \mid f_{ij} \notin Obs \wedge Ch_{ij} \in Fam_i \} \right|$$

Of course, the set ranking sr_i captures the effectiveness of a family to produce new observations, which, in turn is directly correlated with the side-channel leakage.

Finally, the overall ranking r_i of an individual is the sum of all the rankings presented in the preceding, i.e., $r_i = er_i + fr_i + sr_i$. Intuitively, such an overall ranking accounts for the diversity resulting from the individual as well as the children produced by it.

Elite selection stage

The role of the elite selection stage in a genetic algorithm is to save remarkable individuals for breeding in future generations. Remarkable individuals have properties and genes that are useful for reaching the objective.

In order to drive our algorithm towards diversity, we pick the individuals based on the following aspects: overall diversity, fitness diversity, fitness value isolation, and fitness value rarity.

Overall diversity. The individuals with genes that produce diverse fitness values are relevant for keeping in future generations. We prioritize selection of such individuals. The overall ranking r_i provides a metric for selecting the individuals that have genes which lead to diverse families. We use the overall ranking r_i to order all the individuals in a population. We pick the top individuals (i.e. with the most potential for diversity) and save them for the next iteration. We call this the elite selection step in Figure 6.

Fitness diversity. Only choosing the individuals with the top overall ranking might lead the search in the state space to a single direction. We want to make sure that the fitness values of the elite individuals will not lead to an excessively limited exploration fo the search space. On this step, we keep the ranking r_i, for the remaining individuals. However, we pick the top individuals that also have the fitness values different from the ones of the individuals from the elite selection step. Since this step is trying to avoid searching in only one direction and the fitness values are different from the ones of the elite selection step, we call this the anti elite selection step in Figure 6.

Fitness value isolation. In a genetic algorithm, a mutation changes slightly the chromosomes of an individual usually leading to a slightly different observed fitness value. We employ such single mutations for family generation as explained earlier. For this purpose, it is useful to pick suitable individuals that make use of such a mutation. Such individuals are the ones for which the neighboring fitness values are not yet found in Obs. This maximizes the potential that a mutation might lead to a new fitness value, that is the neighbor of the mutated individual's fitness value. Therefore, we rank the remaining individuals in the population, after the elite and anti elite steps, based on how many neighboring fitness values are already in Obs. We consider the neighbors at a distance of one to the left and right. Higher distances could be considered. The individuals with fewer neighbors in Obs are ranked higher. This step is called the neighborless elite step in Figure 6.

Fitness value rarity. During the execution of `GDivAn_C`, fitness values can appear multiple times. The observed fitness values that appeared a large amount of times are not of high interest. We are, however, interested in fitness values that are rare. Individuals that produce such values might have interesting genes we want to save for future generations. Therefore, we rank the remaining individuals from the population based on the rarity of their fitness. We pick the top individuals with the rarest observed fitness values (lowest values in the fitness appearance histogram). We refer to this selection as the low histogram elite selection in Figure 6.

We consider that fitness value isolation is more important than fitness rarity when choosing elite individuals. Lets consider two individuals Ind_i and Ind_j with observed fitness values f_i and f_j, respectively. We consider that f_i has appeared only once in Obs, but f_i-1 and f_i+1 exist as well in Obs. Now, we consider f_j has appeared a thousand times in Obs, during the execution of `GDivAn_C`. However, $f_j - 1$ and $f_j + 1$ do not yet exist in Obs. Picking individual Ind_j over Ind_i gives higher probability to find the neighboring values of f_j. Therefore, we prioritize the selection of individuals with isolated fitness values over the ones with rare fitness values.

It is important to mention that some of the selection steps might be redundant in some generations. For example, the most diverse individuals could have the rarest fitness values. Or that individuals selected in the anti elite selection step could have the top isolated fitness values. We still require all the elite selection steps to be present, as a safety mechanism in case some important individuals are not selected in the previous steps.

Children selection

The elite selection stage is useful for keeping important genes in the population. The children selection stage helps the genetic algorithm with the diversification of the search. The children are ranked based on the rarity of their fitness value. Children with rare fitness values are more likely to lead to new observed values. Therefore, for the new population, after the elite individuals have been selected, the rest of the population for the next generation is picked from the top ranked children.

We do not consider fitness isolation in the children selection stage, as that is the role of the elite selection steps. Since the children are useful for driving the search, rare (new) values are always more valuable than existing values. Children selected will have the possibility to be kept in future generations in the elite selection stage, where the fitness isolation will be taken into account.

If, after a specified number of iterations, the number of found values in Obs doesn't increase, we generate another random batch of individuals and we place them instead of the children in the population for the next iteration. The analysis will terminate after a predefined number of iterations.

`GDivAn` limitations for side-channel analysis

Our proposed solution for side-channel analysis, `GDivAn_C`, is a measurement based approach. Therefore, we cannot provide a guarantee that the observed number of distinct values is the maximum achievable value. However, such a limitation is fundamental and unavoidable for any approach based on testing and measurement.

6. Evaluation

6.1. Evaluation of `GDivAn_T` to obtain WCET

Experimental setup. We use GKLEE [32] as the symbolic execution tool for `GDivAn_T`. GKLEE is a symbolic analyzer and test generator tool tailored for CUDA C++ programs. We modify the source code of GKLEE to obtain the relevant information (e.g. branch divergence, number of instructions) for each explored path and drive the genetic algorithm stage within `GDivAn_T`.

We have picked four kernels for evaluating `GDivAn_T`. Specifically, we have chosen kernels involving multiple program paths to stress-test the mechanism implemented within `GDivAn_T`. Table 3 captures some salient properties of the chosen subject programs. `NSICHNEU` is a single-threaded CPU program obtained from Mälardalen WCET benchmarks [33]. `NSICHNEU` exhibits complex control flow and implements the simulation of a Petri net. We have modified it to a multi-threaded CUDA program, where each thread of the GPU runs the simulation of a

Petri net. LBM is a GPU program for computational fluid dynamics using Lattice Boltzmann Models. BFS is the GPU implementation of breadth-first search and it is obtained from the Rodinia 3.1 benchmark suite [34]. Convolution is a GPU program that applies a convolution filter to the kernel input, based on the values of the respective input.

Program name	#Kernels	#lines of code	#Kernel invocations	#if stmts.	#loops	#threads
LBM	1	97	1	13	0	32768
BFS	2	17	>1	2	1	1024
		11	>1	1	0	1024
NSICHNEU	1	2346	1	252	0	4096
Convolution	1	17	1	2	2	262144

Table 3: Kernel properties

All the kernels have been evaluated on an NVIDIA Tegra K1 GPU. The kernels were compiled with CUDA nvcc version 6.5. For measuring the execution time on the Tegra K1, the default frequencies have been used, i.e., 72 MHz for the GPU's core clock and 204 MHz for the GPU's memory clock. Each generated test was executed ten times in the GPU and the averages of these ten runs are reported in the evaluation.

Evaluating the hypothesis of GDivAn_T. In order to evaluate the key hypothesis of GDivAn_T, we have implemented a synthetic kernel as shown in Listing 1 to run with 2^{15} threads. This kernel is similar to the example in Figure 1(b) where branch divergence does *not* lead to the WCET of the kernel. In this section, we will refer to this kernel as Artificial.

```
__global__ void kernel(int *values, int *result){

  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  if (values[tid] == CONSTANT){
    atomicAdd(&result[blockIdx.x], values[tid]);
    ...
    atomicAdd(&result[blockIdx.x], values[tid]);
  }else{
    atomicAdd(&result[tid], 10);
  }

}
```

Listing 1: An example to test the hypothesis of GDivAn_T

Symbolic execution stage.

Table 4 captures the maximum number of threads GKLEE can run within an hour, the branch coverage obtained and the total time taken to symbolically execute the respective programs. Recall (see Section 5.1.1) that our goal is to achieve a branch coverage as close as possible to 100%. For `Artificial` this has been achieved with two threads and the execution time is 1 second. For the other benchmarks, the branch coverage and execution time is indicated with the maximum number of threads that could be analyzed in less than an hour. We have to mention here that the number of threads to be considered is different for each benchmark. `LBM`, for example, works on square matrices and each element is assigned to one thread. Thus, the next level after four threads would be nine which, however, lead to an analysis time beyond one hour. Similar considerations apply to `BFS` and `NSICHNEU`.

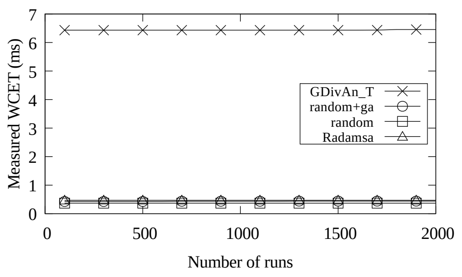
Recall that we use the paths explored by GKLEE to create individuals for the genetic algorithm (Section 5.1.2).

Program	# threads	branch coverage (%)	execution time (secs)
`Artificial`	2	100	1
`LBM`	4	80	10
`BFS`	2	83.33	1
`NSICHNEU`	2	70.24	341
`Convolution`	8	100	11

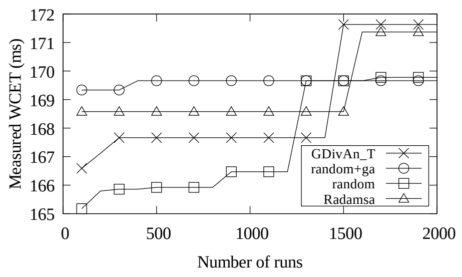
Table 4: Evaluation of GKLEE runs with our subject programs

Genetic algorithm stage. We select a population size of 100 (i.e. number of inputs in one generation) to run our genetic algorithm. Moreover, we keep the elite percentage 10% (cf. Section 5.1.3) and the probability to mutate a gene (for a given individual) to be 25%. It is worthwhile to note that we mutate only one gene of an individual, thus keeping the overall mutation rate (across all genes of the individuals) quite low. Finally, while creating the initial population of individuals, we reserve 30% population for individuals with special traits. These special individuals were created from paths that exhibited maximum number of instructions and branch divergence during the symbolic execution. The parameters of the GA have been set after a preliminary set of extensive experiments.

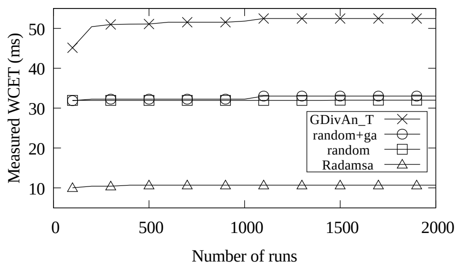
Overall evaluation. Figure 7 outlines the overall evaluation of `GDivAn_T`. To stress test our approach, we compare `GDivAn_T` with random testing ("random" in Figure 7), our genetic algorithm without the symbolic execution step ("random+ga" in Figure 7), and a state-of-the-art fuzz testing tool, namely Radamsa [6]. For "random+ga" approach, we created the initial population of genetic algorithm randomly. Radamsa is a black-box fuzzer. We provide random samples to Radamsa for each target program. Radamsa uses the respective samples to mutate and generate input tests. For a fair comparison, the number of test runs across all approaches is kept the same. Figure 7 clearly shows that the `GDivAn_T` approach outperforms the rest in terms of exposing the kernel WCET.



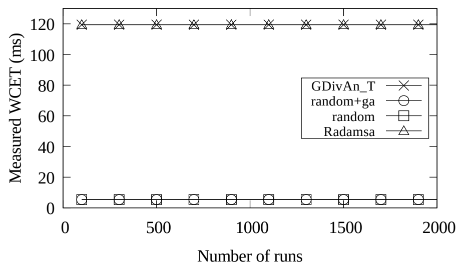
(a) Artificial



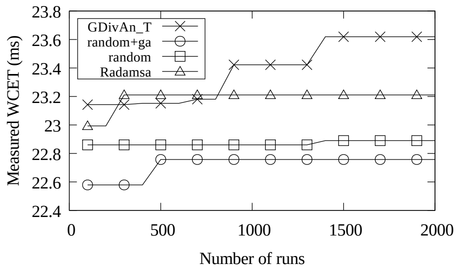
(b) LBM



(c) BFS



(d) NSICHNEU



(e) Convolution

Figure 7: WCET of kernels via Random, Random+GA, Radamsa and `GDivAn_T`

Table 5 captures the WCET detected with the four approaches. The quality of WCETs obtained via `GDivAn_T` is significantly higher than that produced via "random" and "random+ga". Only in the case of the program `NSICHNEU`, Radamsa performs as good as our GDivAn_T approach. It is worthwhile to note that our approach does not always produce the best quality WCET from the start. This is noticeable for LBM and Convolution where GDivAn_T requires some generations to converge to a higher measured WCET than the other three approaches. Table 5 also highlights the time consumed for each approach until it reached the WCET it was able to produce (after that time no improvements on the measured WCET were observed).

	GDivAn_T		random+ga		random		Radamsa	
Program name	WCET (ms)	WCET reached after (s)	WCET (ms)	WCET reached after (s)	WCET (ms)	WCET reached after (s)	WCET (ms)	WCET reached after (s)
Artificial	6.454	459	0.44	258	0.369	1	0.469	1.4
LBM	171.631	12028	169.661	1514	169.777	7255	171.369	7083
BFS	52.482	1411	33.03	911	32	1128	10.679	239
NSICHNEU	119.412	12669	5.419	13333	5.381	2702	119.419	6790
Convolution	23.619	719	22.757	253	22.889	742	23.21	197

Table 5: Testing time. All experiments were performed on an Intel i5 machine having 16GB RAM and running Ubuntu 16.04

The analysis time reported in Table 5 is the sum of the GA steps, the time for data allocation and copying on the GPU, and the kernel execution time. The kernel execution time takes from around 10% (for NSICHNEU) to around 33% (for BFS) of the total analysis time, until GDivAn_T reports the WCET of the respective programs. Note that this kernel execution time already accounts running the respective kernel ten times for each input. For GDivAn_T, the indicated time also includes the duration of the symbolic execution (see Table 4).

6.2. Evaluation of GDivAn_C to test side-channel leakage

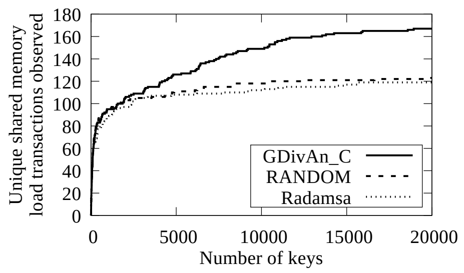
Experimental setup. We use several cryptographic programs implemented in CUDA to test the efficacy of GDivAn_C. Specifically, we choose AES, Blowfish, Camellia and CAST5 to evaluate the effectiveness of GDivAn_C. The GPU implementations of the respective subject algorithms have been selected from [35] and [36]. The implementations selected from [35] are GPU versions of the respective OpenSSL implementations. Thus, we refer to these implementations as *OpenSSL implementations*. The implementations selected from [36] are part of the ISPASS 2009 benchmark suite. We refer to these implementations as *ISPASS implementations*. All our chosen implementations are encryption routines. Table 6 summarizes the implementations chosen for evaluation.

Algorithm name	Origin	Key size (bits)	Reference name
AES	ISPASS	128	ISPASS_AES128
AES	ISPASS	256	ISPASS_AES256
AES	OpenSSL	128	OpenSSL_AES128
AES	OpenSSL	256	OpenSSL_AES256
Blowfish	OpenSSL	128	OpenSSL_BF
Camellia	OpenSSL	128	OpenSSL_Camellia
CAST5	OpenSSL	128	OpenSSL_CAST5

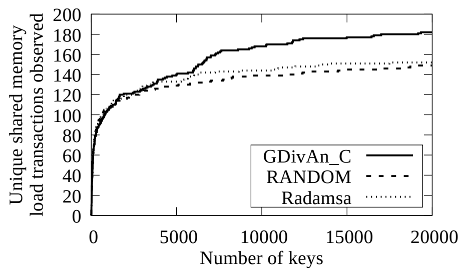
Table 6: Selected algorithms

We have tested each of the implementations from Table 6 with our `GDivAn_C` approach and we have compared the effectiveness of `GDivAn_C` against a testing strategy that randomly generates secret keys for the encryption routine and the test input fuzzer Radamsa. As in the case of the WCET analysis evaluation, we have provided Radamsa with random input key samples from which it could generate keys for test purposes. The target of our experiments was to expose as many side-channel observations as possible, as the number of side-channel observations can be used to quantify the side-channel information leakage. Specifically, for a given input message, we observe the number of shared memory transactions with respect to the secret key.

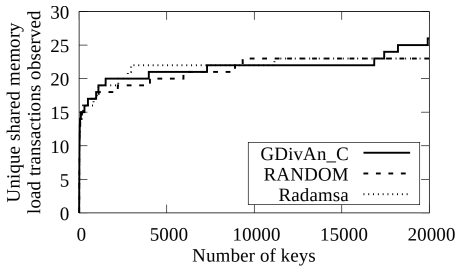
Figure 8 captures the results of the comparison between random key generation , keys generated by Radamsa and generating keys via `GDivAn_C`. We observe that `GDivAn_C` explores the search space of keys in a more efficient manner. Specifically, the number of unique observations exposed by `GDivAn_C` is higher than the observations found through random key generation or by keys generated using Radamsa. We also observe that the OpenSSL implementations do not exhibit a large number of unique observations as compared to the respective ISPASS imple-



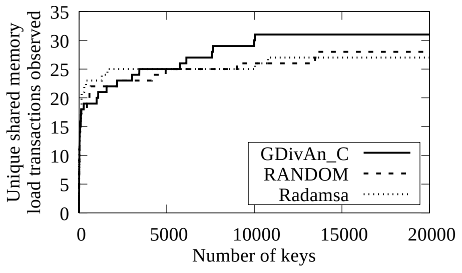
(a) ISPASS_AES128



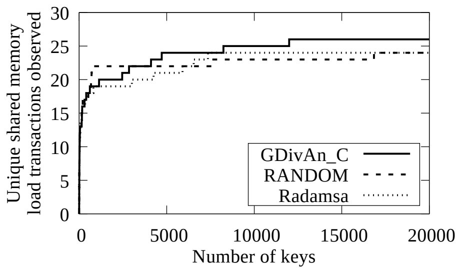
(b) ISPASS_AES256



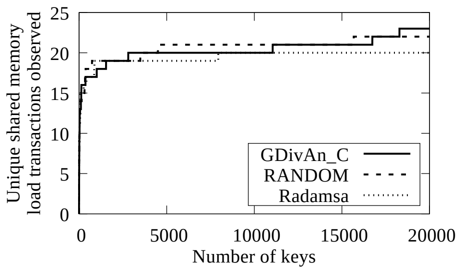
(c) OpenSSL_AES128



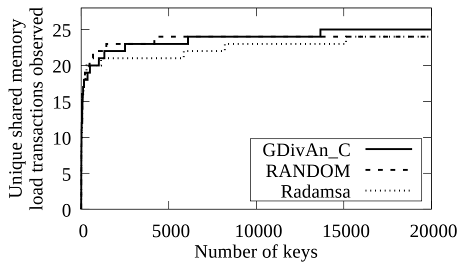
(d) OpenSSL_AES256



(e) OpenSSL_BF



(f) OpenSSL_Camellia



(g) OpenSSL_CAST5

Figure 8: Comparing the number of unique observations when using random generated keys and a fuzzer versus `GDivAn_C`

mentations of the same algorithm. `GDivAn_C` is sometimes outperformed by the random approach or by Radamsa during the testing of the OpenSSL implementations. Even though for a smaller number of keys, `GDivAn_C` does not always perform better, as the number of tested keys increases, `GDivAn_C` provides a higher number of distinct observations.

The efficacy of `GDivAn_C` is better underlined in the ISPASS versions, where it is clear that `GDivAn_C` finds up to 35% more observations as compared to random testing and up to 40% more observations than the fuzzer Radamsa, as seen

	Unique observations			% increase by using GDivAn_C	
Reference name	RANDOM	Radamsa	GDivAn_C	vs. RANDOM	vs. Radamsa
ISPASS_AES128	123	119	167	35.7%	40.3%
ISPASS_AES256	149	152	182	22.1%	19.7%
OpenSSL_AES128	23	23	26	13%	13%
OpenSSL_AES256	28	27	31	10.7%	14.8%
OpenSSL_BF	24	24	26	8.3%	8.3%
OpenSSL_Camellia	22	20	23	4.5%	15%
OpenSSL_CAST5	24	24	25	4.1%	4.1%

Table 7: Comparison between RANDOM, Radamsa and `GDivAn_C`. All experiments were performed on an Intel i5 machine having 16GB RAM and running Ubuntu 16.04

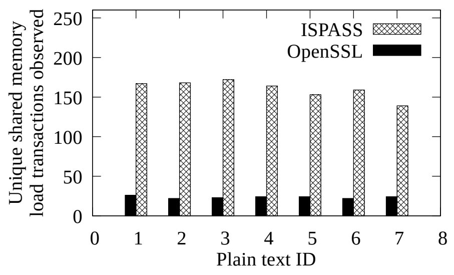
in Table 7. The results from Table 7 have been obtained on an Nvidia GeForce GTX 1060M, using CUDA 8.0. We have used the NVIDIA `nvprof` to profile and obtain the required metrics from the execution of the listed GPU programs.

Comparing robustness of implementations w.r.t. side-channel leakage. Our approach `GDivAn_C` can be used to rank the robustness of different implementations of the same algorithm. In particular, finding more unique observations for an implementation of an algorithm leads to more information being leaked by that implementation and, thus, reduced robustness. We observe from the results of Table 7 that the ISPASS implementations for AES are less robust than the OpenSSL implementations for AES. The results from Table 7 were tested on the same 16 byte plain text. Since the plain text might have an influence on the possible number of observations, it might be argued that this observation is only valid for that single plain text and does not fairly characterize the implementation. Therefore, we have also evaluated the robustness of OpenSSL and ISPASS implementations with different plain texts.

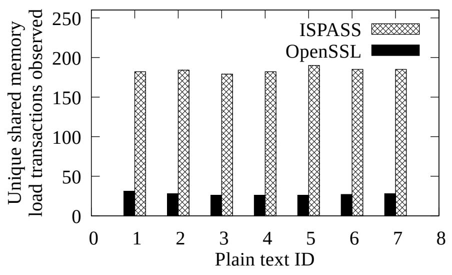
Figure 9 contains the observations gathered by `GDivAn_C` when running AES for different plain texts. We observe that the OpenSSL implementation consistently remains more robust than the ISPASS implementation even for different plain texts. These results show the utility of `GDivAn_C` to compare the robustness of different implementations with respect to side-channel leakage and the consistency of the results obtained.

7. Conclusion and discussion

In this paper, we propose `GDivAn`, a framework for estimating non-functional properties of GPU programs. In its core, `GDivAn` embodies a novel approach



(a) AES128



(b) AES256

Figure 9: Comparing the robustness of different AES implementations (from OpenSSL and ISPASS benchmark suite) when using `GDivAn_C` with different plain texts

based on genetic algorithm driven estimation. To test the WCET of arbitrary GPGPU programs, `GDivAn` systematically combines the strength of symbolic execution and genetic algorithm to converge towards the WCET. We evaluate this approach with several GPU kernels and show its effectiveness compared to random testing and genetic algorithm in their pure forms, and a state-of-the-art fuzz testing tool, namely Radamsa. In the future, the capability of `GDivAn` can be extended to compute the response time of arbitrary GPU-based applications represented as task graphs.

We further employ `GDivAn`, with its evolutionary heuristic based on genetic algorithms, for testing the side-channel leakage of GPGPU programs. More specifically, we investigate the leakage through shared-memory access information for GPU implementations of cryptographic algorithms. The results from comparing against a random exploration approach and the fuzz testing tool Radamsa show that `GDivAn` can expose more side-channel information. As shown in the evaluation, `GDivAn` is also useful for ranking different implementations of the same algorithm with respect to the leakage via side channel.

Acknowledgements

This work is partially supported by the Swedish Research Council (VR) under the grant NT-2017-04194 and partially supported by the Ministry of Education of Singapore under the grant MOE2018-T2-1-098.

References

[1] A. Horga, S. Chattopadhyay, P. Eles, Z. Peng, Measurement based execution time analysis of GPGPU programs via SE+GA, in: 2018 21st Euromicro

Conference on Digital System Design (DSD), IEEE, 2018, pp. 30–37.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al., An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al., The worst-case execution-time problem–overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems (TECS) 7 (3) (2008) 36.

[4] J. Rosen, A. Andrei, P. Eles, Z. Peng, Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip, in: RTSS, 2007, pp. 49–60.

[5] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, H. Falk, A unified WCET analysis framework for multicore platforms, ACM Trans. Embedded Comput. Syst. 13 (4s) (2014) 124:1–124:29.

[6] Radamsa (2016).
URL `https://github.com/akihe/radamsa`

[7] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. C. Berg, S. Wang, An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads, in: RTAS, 2017, pp. 353–364.

[8] G. A. Elliott, B. C. Ward, J. H. Anderson, GPUSync: A framework for real-time GPU management, in: RTSS, 2013, pp. 33–44.

[9] K. Berezovskyi, L. Santinelli, K. Bletsas, E. Tovar, WCET measurement-based and extreme value theory characterisation of CUDA kernels, in: RTNS, 2014, pp. 279:279–279:288.

[10] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, E. Tovar, Measurement-based probabilistic timing analysis for graphics processor units, in: ARCS, 2016, pp. 223–236.

[11] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, et al., Proartis: Probabilistically analyzable real-time systems, ACM Transactions on Embedded Computing Systems (TECS) 12 (2s) (2013) 94.

[12] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, L. Cucu-Grosjean, Open challenges for probabilistic measurement-based worst-case execution time, IEEE Embedded Systems Letters 9 (3) (2017) 69–72.

[13] L. Santinelli, J. Morio, G. Dufour, D. Jacquemart, On the sustainability of the extreme value theory for WCET estimation, in: OASIcs-OpenAccess Series in Informatics, Vol. 39, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[14] G. Lima, D. Dias, E. Barros, Extreme value theory for estimating task execution time bounds: A careful look, in: ECRTS, 2016, pp. 200–211.

[15] A. Betts, A. Donaldson, Estimating the WCET of GPU-accelerated applications using hybrid analysis, in: Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on, IEEE, 2013, pp. 193–202.

[16] A. Marref, Evolutionary techniques for parametric wcet analysis, in: 12th International Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[17] M. Tlili, S. Wappler, H. Sthamer, Improving evolutionary real-time testing, in: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM, 2006, pp. 1917–1924.

[18] I. Ashraf, G. M. Hassan, K. Yahya, S. A. Shah, S. Ullah, A. Manzoor, M. Murad, Parameter tuning of evolutionary algorithm by meta-eas for wcet analysis, in: 2010 6th International Conference on Emerging Technologies (ICET), IEEE, 2010, pp. 7–10.

[19] J. P. Galeotti, G. Fraser, A. Arcuri, Improving search-based test suite generation with dynamic symbolic execution, in: 2013 ieee 24th international symposium on software reliability engineering (issre), IEEE, 2013, pp. 360–369.

[20] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, T. E. J. Vos, Symbolic search-based testing, in: ASE, 2011, pp. 53–62.

[21] Z. H. Jiang, Y. Fei, D. Kaeli, A novel side-channel timing attack on GPUs, in: Proceedings of the on Great Lakes Symposium on VLSI 2017, ACM, 2017, pp. 167–172.

[22] Z. H. Jiang, Y. Fei, D. Kaeli, A complete key recovery timing attack on a GPU, in: 2016 IEEE International symposium on high performance computer architecture (HPCA), IEEE, 2016, pp. 394–405.

[23] C. Luo, Y. Fei, P. Luo, S. Mukherjee, D. Kaeli, Side-channel power analysis of a GPU AES implementation, in: 2015 33rd IEEE International Conference on Computer Design (ICCD), IEEE, 2015, pp. 281–288.

[24] R. Di Pietro, F. Lombardi, A. Villani, CUDA leaks: a detailed hack for CUDA and a (partial) fix, ACM Transactions on Embedded Computing Systems (TECS) 15 (1) (2016) 15.

[25] CUDA toolkit documentation (2017).
URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[26] B. Köpf, L. Mauborgne, M. Ochoa, Automatic quantification of cache side-channels, in: International Conference on Computer Aided Verification, Springer, 2012, pp. 564–580.

[27] T. Basu, S. Chattopadhyay, Testing cache side-channel leakage, in: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2017, pp. 51–60.

[28] G. Doychev, B. Köpf, L. Mauborgne, J. Reineke, Cacheaudit: A tool for the static analysis of cache side channels, ACM Transactions on Information and System Security (TISSEC) 18 (1) (2015) 4.

[29] C. S. Pasareanu, Q.-S. Phan, P. Malacaria, Multi-run side-channel analysis using Symbolic Execution and Max-SMT, in: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), IEEE, 2016, pp. 387–400.

[30] J. Demme, R. Martin, A. Waksman, S. Sethumadhavan, Side-channel vulnerability factor: A metric for measuring information leakage, in: 2012 39th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2012, pp. 106–117.

[31] E.-G. Talbi, Metaheuristics: from design to implementation, Vol. 74, John Wiley & Sons, 2009.

[32] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. P. Rajan, GKLEE: concolic verification and test generation for GPUs, in: PPOPP, 2012, pp. 215–224.

[33] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, The Mälardalen WCET benchmarks – past, present and future, in: WCET, 2010, pp. 137–147.

[34] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IISWC, 2009, pp. 44–54.

[35] J. Gilger, J. Barnickel, U. Meyer, GPU-acceleration of block ciphers in the openssl cryptographic library, in: D. Gollmann, F. C. Freiling (Eds.), Information Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 338–353.

[36] S. A. Manavski, CUDA compatible GPU as an efficient hardware accelerator for AES cryptography, in: 2007 IEEE International Conference on Signal Processing and Communications, IEEE, 2007, pp. 65–68.