CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World

Jonas Roels
jonas.roels@kuleuven.be
Distrinet, KU Leuven
Ghent, Belgium

Adriaan Jacobs
adriaan.jacobs@kuleuven.be
Distrinet, KU Leuven
Ghent, Belgium

Stijn Volckaert
stijn.volckaert@kuleuven.be
Distrinet, KU Leuven
Ghent, Belgium

Abstract

The rising popularity of Graphics Processing Units (GPUs) has made them an attractive target for attackers looking to steal Intellectual Property (IP) such as ML models or disrupt the operation of heterogeneous computing systems. However, defending against GPU attacks is anything but trivial since the inner workings of these–often proprietary–devices are still poorly understood. Preliminary work demonstrates a worrying similarity to the attack surface of the CPU domain, particularly concerning the memory unsafety of device-side code. We corroborate these worrying findings by constructing the first rigorous experimental analysis of input-triggered, ROP-based exploits entirely within device-side NVIDIA CUDA code. We repurposed known CPU-based code-reuse attack techniques to unlock previously unusable gadgets in this code and demonstrate that the gadget set is Turing-complete, enabling attackers to perform arbitrary computations. We conclude that ROP attacks on GPUs are feasible and more potent than previously thought.

Following this discovery, we evaluate the strength of current device-side mitigations, such as stack canaries and Address Space Layout Randomization (ASLR). Given the lack of more powerful protection mechanisms, these basic security measures play a crucial role in GPU security. However, we find them even less secure than their CPU counterparts. Our findings indicate that the GPU domain urgently needs robust protection mechanisms that fit the unique GPU architectures and address the flaws in existing systems.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

Graphics Processing Unit, GPU, NVIDIA, CUDA, Memory Safety, Return-Oriented Programming, ROP, Code-Reuse Attacks

ACM Reference Format:
Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *The 18th European Workshop on Systems Security (EuroSec'25), March 30-April 3 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3722041.3723099

EuroSec'25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1563-1/2025/03
https://doi.org/10.1145/3722041.3723099

1 Introduction

In recent years, the popularity of Graphics Processing Units (GPUs) has been on the rise. These accelerator cards contain highly-parallel circuitry that improves the speed and power efficiency of an ever-increasing number of software applications, ranging from traditional graphics rendering to Artificial Intelligence (AI) [1], Machine Learning (ML) [2, 3], simulation [4], and graph processing [5].

Following the rapid adoption of GPUs across the software landscape, applications using GPU acceleration have become increasingly attractive targets for malicious actors to exploit. They often handle sensitive data or hold important decision-making power, e.g., in data encryption or ML inference applications, while being entrusted to *peripheral* hardware platforms that have historically not been exposed to or hardened against malicious users [6–8]. Leveraging the inadequacies of these platforms, attackers have been able to exploit GPU-accelerated applications for various reasons; to leak sensitive data [9–11] such as encryption keys [12–14], keystrokes [15, 16], rendered web pages [17–20], or ML model weights [17, 21–23], to influence ML training and inference [17, 24, 25], to hijack or waste computing power [26, 27], and to break out of the GPU environment to attack the host system itself [6, 8].

While many of these attacks have focused on breaking the isolation between users in a multi-tenant device-sharing threat model [17, 28–31], like in the cloud [32], alternative avenues of exploitation also exist. GPUs are generally programmed using dialects of C++ [33, 34], and, as such, inherit many of the same memory safety issues that have plagued C/C++ applications on the CPU for decades [35]. Previous work already showed that, despite the differences in architecture [25], memory errors on accelerator hardware can once again be used to gain arbitrary read and write primitives [36, 37], which grants attackers similar power over the address space as on the CPU [38]. For instance, researchers were able to launch Return-Oriented Programming (ROP) [39] attacks [25, 38], to change the weights of an ML model or to overwrite code pages in the NVIDIA CUDA programming environment. While these attacks highlight the possibility of ROP on CUDA, they do not demonstrate its full potential: achieving arbitrary computation independently, a feat proven on the CPU over a decade ago [40, 41]. Previous work suggests that the applicability of ROP on GPUs is limited by the scarcity of available gadgets, a result of the alignment of the Program Counter (PC), which makes discovering unintended gadgets challenging [38], nay impossible.

This paper demonstrates that using ROP on GPUs allows for arbitrary computation by providing the **first rigorous experimental analysis of a Turing-complete, input-triggered, ROP-based weird machine entirely within device-side NVIDIA CUDA code**. Additionally, similar to return-to-libc style attacks on the CPU [42], this ROP chain exclusively utilizes gadgets found in the CUDA runtime library, which serves as the CUDA equivalent of

the *standard C library* (*libc*) on the CPU, making them accessible in every CUDA program. We also demonstrate that widely used CUDA libraries contain ROP gadgets, which enhance the attacker's capabilities and reduce their reliance on the runtime library. We achieve this using a technique we call *Compounding ROP Gadgets*, which borrows ideas from previous work on code-reuse attacks on the CPU [43–45], that enable the exploitation of gadgets dismissed by previous work [38]. To build the tools necessary for creating this gadget set, **we further reverse-engineered the Turing SASS ISA** to refine the insights provided by previous work [46, 47]. In this way, we demonstrate that code-reuse attacks are feasible on GPUs and more potent than previously thought.

Our findings reinforce the growing awareness that GPUs require strong protection mechanisms against such expressive attacks. However, we find that the two most commonly used and accessible protection mechanisms on GPUs—stack canaries and Address Space Layout Randomization (ASLR) [48]—have weaknesses that leave them vulnerable to exploits. In particular, **we discovered two weaknesses in the implementation of stack canaries by the NVIDIA CUDA Compiler (NVCC)** [49]: it fails to insert canaries into all vulnerable functions, and due to weak error propagation between the GPU and CPU, a canary check failure does not terminate the host process. This, combined with the fact that canaries on the GPU are only randomized at process startup, makes them susceptible to leakage and brute-force attacks. Even though canaries are much more essential on the GPU due to the lack of other basic protection mechanisms, such as Data Execution Prevention (DEP) [38, 50], they are even less secure than on the CPU.

Like stack canaries, **ASLR is susceptible to leakage and brute-force attacks**, as its randomization occurs only at process launch. Thus, we conclude that both mitigations can be easily bypassed and will not stop determined adversaries from launching input-triggered ROP attacks on the GPU.

Finally, we find that currently, **no existing GPU-based exploit mitigation** [51–56] **or sanitizer** [57–63] **offers strong security guarantees while also being attractive to deploy**. Accordingly, we would like this paper to serve as a call to action for the GPU industry and the academic world to develop new and improved protection mechanisms.

2 The CUDA GPU Architecture

This section describes the CUDA GPU architecture, more specifically, we focus on the Turing microarchitecture in this paper. CUDA programs consist of host and device code [49], which the host can invoke through special functions called CUDA kernels. These kernels take additional parameters, namely the number of blocks and threads per block. The CUDA runtime divides these blocks into warps, typically containing at most 32 threads each, and schedules those over the available Streaming Multiprocessors (SMs).

The host communicates with the NVIDIA driver through the CUDA runtime library to initiate kernels and interact with the GPU in other ways, such as allocating or copying memory. This library provides the functions necessary for these operations and performs the required system calls to communicate with the GPU. Next to the host code, this library also contains device code, which provides functions that would be provided by the Operating System's (OS)

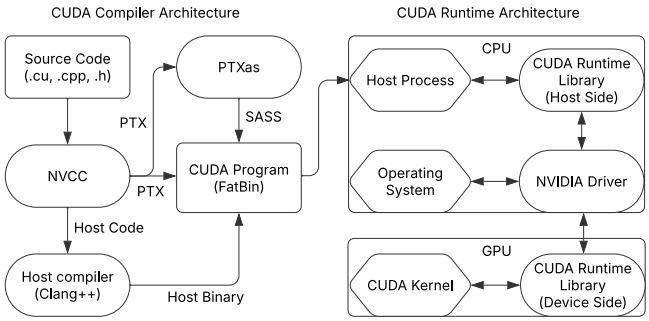


Figure 1: The CUDA Compiler and Runtime Architecture

kernel or *libc* on the CPU. Examples of these are memory allocation functions, e.g., `malloc` and `free`, and I/O operations, such as `printf`. Figure 1 shows the interaction between the CUDA kernel, host process, and driver using the runtime library.

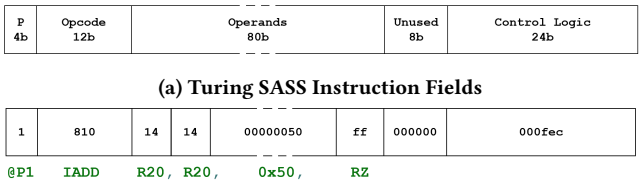
While executing, CUDA kernels have access to various memory spaces [33], namely: global memory, which serves as the heap and is used for storing code and exchanging data with the host; local memory, which forms a thread local stack; shared memory, which is a fast, on-chip memory for data-exchange between threads in the same block; and constant memory, a fast, read-only memory. For each of these memory spaces, CUDA provides specific instructions, e.g., `LDL/STL` for local memory, and generic ones, e.g., `LD/ST`, that are capable of accessing global, local, and shared memory [38].

CUDA GPUs also have access to various registers, including 256 thread-local general-purpose registers, 64 warp-local uniform registers, 8 thread-local predication registers [46], 12 thread-local barrier registers, and special registers, e.g., containing information about the location of the thread within the block.

The NVIDIA CUDA Compiler (NVCC) can produce device-side binaries in two distinct formats [49]: Parallel Thread Execution (PTX) and Streaming Assembly (SASS). PTX is an intermediate representation of the program that is Just-in-Time (JIT) compiled by the CUDA runtime into SASS for the correct architecture, which provides portability across different CUDA devices. In contrast, SASS is the architecture-specific machine code directly executed on CUDA-enabled GPUs. The compilation from PTX to SASS happens using the *PTXas* compiler backend. Furthermore, NVCC can also produce fatbinaries, which are files containing a combination of PTX and SASS code for multiple architectures. In addition, NVCC relies on a C++ compiler such as Clang++ to produce the host binary. Figure 1 visually represents this compiler hierarchy.

NVIDIA only provides limited documentation for the SASS ISA and frequently updates it across GPU microarchitectures [64]. Turing SASS instructions use a 16-byte format that, according to Jia et al. [46], consists roughly of 12 bytes encoding the instruction information, 3 bytes used for control logic, and a single unused byte. The control logic determines how the CUDA runtime schedules operations between SMs and the synchronization and yielding of threads.

Another key aspect of CUDA devices is predicated execution. This pattern makes control-flow divergence among threads possible and uses predication registers to selectively execute instructions



(a) Turing SASS Instruction Fields

(b) Example of a Turing SASS Instruction

Figure 2: Turing SASS Instruction Format.

on specific threads. As a general rule, developers should minimize thread divergence whenever possible to maximize performance, as diverging threads reduce the number of parallelized operations.

.

3 Reverse Engineering the Turing SASS ISA

A key step in crafting ROP gadgets is analyzing memory contents. NVIDIA-provided tools such as *Cuobjdump* and *Nvidisasm* [64] are unsuitable for this purpose, since they expect well-formatted input. Instead, we developed a custom tool that analyzes the memory dump and decodes the binary instructions into human-readable SASS code. To do this, we first needed to gain a deeper understanding of the under-documented SASS ISA, particularly how operands and predicated execution are represented in the opcode. Jia et al. already provide some insight about SASS control logic [46], but do not cover all possible instructions, nor the encoding of their operands. Other work only describes older architectures [47], which NVIDIA has completely overhauled.

Through binary analysis, we discovered that an instruction comprises five parts, as shown in Figure 2a. The instruction starts with a 4-bit predication field (P), whose encoding on the Turing architecture remains unchanged from older architectures analyzed by Hayes et al. [47]. Three bits (r) indicate which predication register to use, with an all-one value (0x7) hard-wired to a special Predication True (PT) value, meaning the instruction is always executed. A 4th bit (n) optionally negates the looked-up predicate. It is nonsensical in conjunction with the PT value since that would constitute a never-executed instruction. We summarize the decoding of the predication field in Equation 1.

$$\text{Predication}(r,n)\begin{cases} \text{P}r & \text{if } r < 7 \text{ and } n == 0 \\ \text{!P}r & \text{if } r < 7 \text{ and } n == 1 \\ \text{PT} & \text{if } r == 7 \end{cases} \quad (1)$$

Next, we find 12 bits that select the opcode, which is followed by ten bytes encoding operands in the form of 32-, 48-, or 64-bit immediates; register of any types; operation modes; or combinations thereof. The general-purpose, barrier, and uniform registers are encoded using a single byte. Predication registers use only 4 bits [47], and are decoded using Equation 1. Operation modes are used in specific instructions to define their precise behavior, such as the number of bits loaded by a load instruction. The final two parts, as described by Jia et al.[46], comprise the control logic and unused bytes. Figure 2b gives an example of an encoded addition instruction, which takes three registers and a 32-bit immediate as operands.

Previous work mentions that to access local memory using generic load/store instructions [38], the CUDA runtime prefixes the offset with a "magic" value to form the address. We found that these magic values take up a single byte, specifically, bits 24 through 31 of the address. Further, we discovered that even when ASLR is disabled, these values vary between different CUDA runtimes, including those on the same system, for example, within a debugging environment such as CUDA-GDB or outside of it. We also found that these values change between driver versions.

4 Threat Model

We consider a heterogeneous application where the host process continuously receives input, launches a CUDA kernel to process this input, and returns the kernel's output, simulating the basic operations of, e.g., a machine learning inference server. We assume that at least one of the device functions invoked by this CUDA kernel contains a memory corruption vulnerability that can be triggered by user input. The host application, in contrast, can be entirely error-free. All parts of the attack occur on the GPU, which we assume has a protection mechanism like Write-XOR-eXecute (W⊕X) in place, making code-injection attacks impossible. While current-generation hardware, including our evaluation platform, does not support this feature [65, 66], future architectures may choose to implement it. Finally, we assume that the GPU has all supported protection mechanisms enabled, such as ASLR and stack canaries.

5 Code-Reuse Attacks on GPUs

This paper focuses on attacking NVIDIA CUDA-enabled GPUs like the NVIDIA Quadro T400. All experiments were conducted on this GPU, utilizing CUDA toolkit version 12.6, NVIDIA driver version 560.35.05, and Clang 18.1.8. The host system comprises an AMD 7800X3D CPU with 64 GB of DDR5 RAM running Ubuntu 24.04.1 LTS with the 6.8.0-51-generic Linux kernel.

5.1 Building a ROP Gadget Set

Traditional ROP gadgets in CUDA code are sequences of instructions that end with a return instruction and include instructions that retrieve the return address from the stack, which is stored in local memory. Unlike x86 CPUs, GPUs do not automatically pop the return address from the stack when executing the return instruction. Instead, they return based on a value held in a register. Furthermore, valid gadgets should not contain any control-flow-altering instructions, such as branches, jumps, or calls, as these could lead to crashes due to the altered program state or exit the ROP chain. Additionally, gadgets containing synchronization instructions must be used with caution. When selecting gadgets, it is important to consider predicated execution, as leveraging this correctly enhances gadget effectiveness at the cost of complexity.

Using tools created by Zhang et al. [65], we dumped the entire GPU memory and extracted the page table contents. Using this information, we determined the physical address of the memory page that contains the device-side CUDA runtime library code. We disassembled this page using a script based on the information outlined in Section 3.

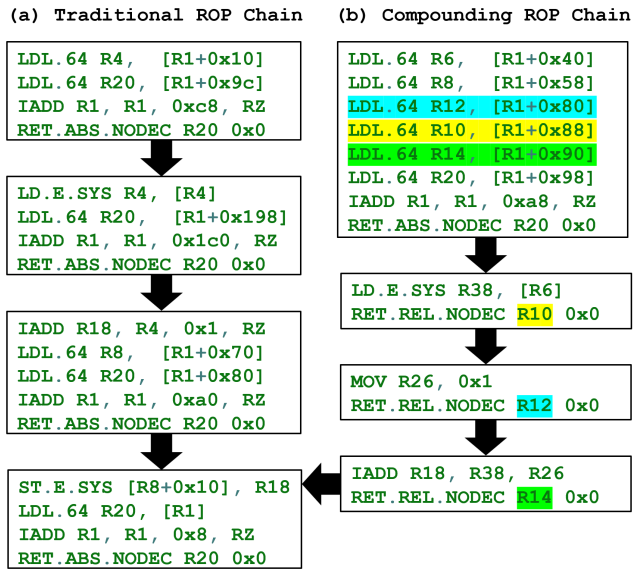


Figure 3: Example GPU ROP chains

Using this process, we discovered 380 return instructions, of which 278 formed valid gadgets according to the constraints outlined above, significantly more than the 52 identified by Guo et al. [38]. Of these, 64 are unique regarding instruction sequence and operands used. Most of these gadgets perform simple operations, such as solely loading registers from the stack. However, we also found more specialized gadgets, including generic load and store operations, atomic operations on generic and global memory, predicate setting operations, and gadgets that overload the stack pointer.

Compounding ROP Chains. By employing a technique similar to those found in advanced ROP attacks on RISC CPUs [44, 45], as well as JOP attacks [43], we discovered additional gadgets, which we call sub-gadgets. These sub-gadgets do not conform to the traditional requirements for ROP gadgets because they do not load the return address from memory. To overcome this limitation, we combine them with a traditional gadget that utilizes a different register for the return address and loads both its and the sub-gadget's return address from the stack. We call this technique *compounding ROP gadgets*. Figure 3 illustrates how traditional and sub-gadgets are combined to form a compounding ROP chain.

This technique has other benefits besides expanding the gadget set, such as using the available stack space more efficiently. Each traditional gadget discovered in the runtime library increments the stack pointer by an average of 86 bytes. On the contrary, the sub-gadgets do not increment the stack pointer but re-purpose otherwise wasted stack space. Many of these sub-gadgets are additionally helpful because they use different registers to implement the same basic operations, which has been found to be a major advantage for solving constraints during gadget selection and minimizing side effects on the CPU [67].

Expressiveness of the Gadget Set. We found that the gadgets required to construct Turing machines and thus perform arbitrary computations are readily available in CUDA kernels. The expressiveness of the resulting gadget chain is limited only by the availability of niche gadgets that the attacker may need to implement their desired exploit. We replicated all operation types of the *MINDOP* language [68], which can be used to craft Turing-complete attacks, using only the gadgets available in the runtime library, which are accessible in every CUDA application. While this language was initially designed to evaluate the expressiveness of Data-Oriented Programming (DOP) [68, 69], we find a similar programming model – where values are written back to memory after every operation – which is also used in ROP Chains targeting RISC CPUs [45], to be well-suited for ROP chains on GPUs as the number of callee-saved registers is often large. Figure 3 shows this behavior. Here, the value is loaded by a gadget, incremented by the next, and stored back in memory by the final gadget. While the large variety of sub-gadgets increases the length of chains that avoid writing to memory, it remains impractical to propagate data between registers solely.

We identified only a limited number of gadgets that directly implemented common arithmetic operations. Specifically, we found one incrementation gadget and one addition gadget. However, we can combine these gadgets with other sequences, particularly conditional jump gadgets, to create sequences that perform other arithmetic operations, such as multiplication and subtraction.

Gadgets in Common CUDA Libraries. Previous work did not identify any gadgets in libraries such as *CuBlas* [70], *CuDNN* [71], and *Torch* [72], due to the absence of functions that push their return address to the stack [38]. Our examination of these libraries showed that they still contained usable sub-gadgets that improve the gadget set's expressiveness. Tensorflow [73], which was not analyzed by previous work, additionally included traditional gadgets. We summarize the gadgets found in each library in Table 1.

While the expressiveness of gadget sets from smaller libraries is often limited, they offer gadgets that enable more efficient ROP chains, requiring less stack space and allowing for more arithmetic operations, such as shift operations. Moreover, larger libraries frequently provide gadgets that can access different GPU memory spaces, which relaxes the need to use generic load/store instructions. Finally, we found that some libraries even include gadgets that overwrite the stack pointer, enabling stack pivoting [74], which forms the basis for conditional jumps between gadgets, loops, and expressive ROP attacks that can perform arbitrary computations. However, the CUDA stack pointer represents an offset from the start of the local memory region and, therefore, can not point to other memory regions, such as the heap, as is common on the CPU.

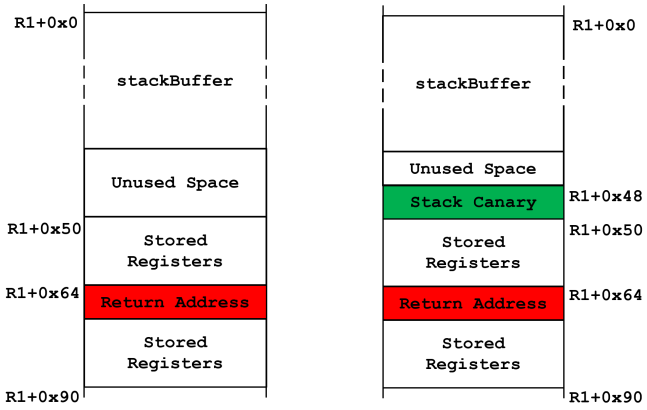
5.2 Exploiting CUDA Applications

The first step in exploiting a CUDA application is to find a function that contains a memory corruption vulnerability. An attacker will exploit this vulnerability to inject their ROP chain into the stack and overwrite a value that will eventually be loaded into the program counter. This can be achieved by overwriting a function pointer or a return address. However, since CUDA programs often do not write their return address to the stack, this process is potentially more difficult than on the CPU. An attacker will analyze this function's memory layout by analyzing the program offline, if available, or using reverse engineering techniques. This involves identifying

Source	Guo et al. [38]	This Paper					
	Total Gadgets	Traditional Gadgets	Sub Gadgets	Total Unique Gadgets	Store Gadgets	Load Gadgets	Stack Pivoting Gadgets
Runtime Library	52	278	8	72	●	●	✓
CuBlas [70]	0	0	34	22	○	○	✗
CuDNN [71]	0	0	1197	315	◓	◓	✗
Torch [72]	N.A.	0	8317	4086	◒	◑	✗
Tensorflow [73]	N.A.	606	9295	4064	●	●	✓

Legend: ○ - None ◴ - Global Memory ◔ - Shared Memory ◒ - Local Memory ● - Generic Memory ✗ - No ✓ - Yes

Table 1: Analysis of Gadget Sets Found in Various CUDA Libraries



(a) Original stack **(b) Stack with canaries**

Figure 4: Stack Layout of the Vulnerable Function

where specific values, such as the return address, are positioned relative to the starting point of the write primitive and identifying the size of the stack frame. Figure 4a displays the memory layout of the vulnerable function in the victim program.

Another requirement is to find the correct "magic" values. These are essential for determining the gadgets' addresses and using local memory. We identified two ways of leaking these. The easiest method is through a buffer overread, which discloses a part of the CUDA executable loaded into memory. Certain opcodes, such as call instructions, contain these "magic" values at runtime.

Another approach is to leak part of the constant memory. In this memory space, the driver loads a data structure resembling the CPU's Global Offset Table (GOT). We use the function pointers in this table to calculate the runtime library code's "magic" value.

Bypassing Common Mitigations. Stack canaries and ASLR have been standard mitigations on the CPU for decades and are also supported on CUDA GPUs. However, despite forming the only line of defense on current-generation GPUs, we find them to be weaker than their CPU counterparts.

In NVCC, stack canaries can be activated using the *device-stack-protector* option and take the form of eight bytes: one byte set to zero and seven randomized bytes. As illustrated in Figure 4b, these bytes are positioned between local variables and callee-saved registers, which includes the return address if spilled. After these registers are saved, NVCC's implementation reads the canary value from constant memory and stores it on the stack. At the end of the function, these bytes are verified against fresh values obtained from constant memory before restoring the register content. We identified two issues with NVCC's implementation of stack canaries.

First, NVCC only inserts stack canaries in functions it deems "high-risk" [49], according to a heuristic in the compiler. However, we successfully used one of these "low-risk" functions to overwrite the return address and subsequently hijack the control flow. We discovered that NVCC does not insert a canary when a function has a relatively small stack frame, specifically when it contains no buffer larger than 40 bytes. To address this issue, we recommend a comprehensive revision of this heuristic, for example, using static analysis to confirm that all memory accesses within a function are safe and only omitting the stack canaries if this condition is met.

The second issue arises from the interplay between error handling and canary re-randomization. When CUDA kernels throw an exception, e.g., due to a failed canary check, they terminate. However, the host process continues executing, and any subsequent kernel this host launches will use the same canary values. This effectively implements a crash-resistant oracle [75], in which the attacker can repeatedly attempt to leak or brute-force the GPU stack canaries. Even though there are 2^{56} possible canary values, given a byte-granular write primitive, an attacker can brute-force the canary value in at most 1792 (7×2^8) tries due to this weakness. We recommend that each CUDA kernel gets its own randomized canary value in constant memory to prevent this exploit. Since launching a new kernel is already a high-overhead operation, we expect the introduced performance impact will be negligible.

Building on previous work [25, 38, 76], we find that ASLR randomizes up to 20 bits of certain virtual addresses through the CUDA runtime on GPUs. Nevertheless, like canaries, these bits are only re-randomized at process launch, making them susceptible to leakage and brute-force attacks. Contrary to previous work [25, 38], we found that ASLR on the GPU also randomizes the addresses of code pages, including those of the driver API. Because CUDA programs compiled without debug symbols often use relative addressing for indirect control-flow transfers, an attacker does not rely on absolute addresses as commonly required on the CPU. Since the current implementation of ASLR on CUDA does not randomize these relative offsets, instead, using a single offset for all code pages, ASLR forms a mere nuisance for attackers.

Preventing Divergences. Approximately one in three gadgets includes a synchronization instruction, such as `BSYNC`, which causes threads to yield and block until all threads within the warp reach the same convergence barriers. Incorrect usage of these gadgets will cause the program to enter a deadlock. We find it impossible to avoid these instructions when crafting expressive ROP chains, as

generally every gadget that operates on memory, such as a store or atomic operation, also executes a synchronization operation before returning. Therefore, an attacker must prevent divergences. This can be achieved by hijacking all threads uniformly, i.e., providing them with the same ROP chain so they perform exactly the same instructions and, thus, never diverge. Divergences could be supported by filling the barrier registers with attacker-controlled values.

Injecting the ROP Chain. To generate the input, an attacker would combine the information outlined in the first part of this section. They would first leverage their knowledge of the memory layout to overwrite the stack canaries with their original valid values and replace the return address with the address of the first gadget. Beyond the original stack frame, the attacker would insert important values for subsequent gadgets, such as the address of the next gadget and values intended for registers. The stack space following this frame limits the length of the ROP chain. However, we find that above the stack used by the kernel and subsequent device functions, a region of 800 bytes is reserved for certain runtime library functions. Since this region is unprotected, an attacker can use it to place their ROP chain. Using stack pivoting techniques [74], the attacker can expand the available space by placing part of the ROP chain elsewhere in local memory, specifically in the buffer's initial portion before the stack canaries and return address.

6 Discussion and Future Work

The available gadgets vary with CUDA toolchain and driver versions. A new gadget set must compile whenever NVIDIA updates the device-side runtime library. Thus, attackers need to know the victim's CUDA version. Changes in this API could allow NVIDIA to redesign it, removing all (powerful) gadgets. This section illustrates how NVIDIA may achieve this and how attackers might bypass it.

Many of our ROP chains, particularly conditional jump sequences, depend significantly on gadgets that load the stack pointer from memory. We find that these gadgets are readily available. Specifically, we identified 104 gadgets in the runtime that can serve this purpose. Consequently, a thorough overhaul of the runtime library would be essential to remove all these instances, as the functions associated with these gadgets play a vital role in executing features usually managed by the OS kernel on the CPU, such as system calls.

Another strategy NVIDIA could consider involves eliminating all traditional gadgets, as an attacker cannot initiate a ROP chain if their gadget set consists entirely of sub-gadgets. While this would hinder attackers from exploiting the runtime library, we demonstrated that, contrary to previous assumptions [38], commonly used libraries contain both traditional ROP gadgets and sub-gadgets, including ones that can facilitate conditional jumps in the ROP chain and access all memory spaces. Which makes expressive ROP chains without using the runtime library possible. Additionally, NVIDIA would have to rework the implementation of recursive functions.

Similar objectives, namely gadget-free compilation, have been proposed on the CPU [77–80]. However, they failed to eliminate all gadgets, opening the door for other techniques, such as return address protection using a shadow or safe stack [81–86], Code-Pointer integrity (CPI) [87–89], and Control-Flow Integrity (CFI) [90–94]. Future work should assess these techniques as they might prove viable for protecting against ROP attacks on GPUs.

Furthermore, future work should focus on developing tools that automatically chain CUDA ROP gadgets together to achieve the attacker's desired outcome. Similar tools exist on the CPU and clearly indicate the threat posed by ROP [67, 95–97], as such they form a clear motivator for developers to build effective protection mechanisms against ROP attacks. Finally, future work should evaluate GPUs from various manufacturers and other types of auXiliary Processing Units (XPUs), as they may share vulnerabilities similar to those of CUDA-enabled NVIDIA GPUs discussed in this paper.

7 Related Work

Research on memory errors on GPUs remains underdeveloped. Foundational works by Di et al. [36] and Miele [37] demonstrated buffer overflows on GPUs, nearly a decade ago. Nevertheless, more recent studies on GPU security focus on aspects such as information leakage between different processes and tenants [12, 14–23].

Recent attacks by Guo et al. and Park et al. rely on the exploitation of memory errors and ROP to corrupt deep learning models using control flow hijacks [25, 38]. While both papers offer valuable insights into memory errors on GPUs, their attacks do not fully demonstrate the true potential of ROP on GPUs and fail to prove that arbitrary computation is possible using ROP. Both use it to construct an arbitrary write primitive to overwrite code pages or corrupt ML model weights. Hence, neither paper demonstrates that ROP can function as a standalone effective attack vector on GPUs.

Defensively, various sanitizers aim to detect GPU memory errors during pre-deployment testing [57–63]. Academics have proposed mitigations to protect GPUs from memory error exploitation [51–56], but none have gained real-world adoption. They typically either slow program execution excessively [57–60], fail to cover the entire GPU memory space [51, 52, 62], neglect non-adjacent Out-of-Bounds accesses [51–53, 57–61], fail to detect errors quickly enough [51, 52], or require hardware modifications, making them incompatible with commodity GPUs [54–56].

8 Conclusion

This paper demonstrates that code-reuse attacks pose a greater threat to GPU systems than previously thought. Using compounding ROP chains, we significantly increase the number of gadgets in the CUDA runtime library and external libraries, drawing inspiration from CPU-based code-reuse attacks. We show that the runtime library and other libraries provide gadgets that can perform arbitrary computations on the victim. Furthermore, we found that the only deployed protection mechanisms, such as ASLR and stack canaries, do not prevent these types of exploits on the GPU. We aim for this paper to serve as a call to action. We find that the GPU space urgently needs a robust protection mechanism that leverages the strengths of GPUs to address the shortcomings of existing systems.

Acknowledgement

This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders. Thank you to the EuroSec '25 the reviewers, for their valuable feedback. We also appreciate our colleagues and proofreaders, particularly Mahmoud Ammar and Jonas Vinck, for their support and discussions during this work's development.

References

[1] NVIDIA Corporation. Nvidia a100 tensor core gpu. https://www.nvidia.com/en-us/data-center/a100/, 2020.

[2] Bill Dally. Hardware for deep learning. *2023 IEEE Hot Chips 35 Symposium (HCS)*, August 2023. URL https://hc2023.hotchips.org/assets/program/conference/day2/Keynote%202/Keynote-NVIDIA_Hardware-for-Deep-Learning.pdf.

[3] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[4] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7. IEEE, 2008.

[5] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, 50 (6), jan 2018. ISSN 0360-0300. doi: 10.1145/3128571. URL https://doi.org/10.1145/3128571.

[6] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete gpus. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 1–11, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349154. doi: 10.1145/3038228.3038233. URL https://doi.org/10.1145/3038228.3038233.

[7] Sparsh Mittal, SB Abhinaya, Manish Reddy, and Irfan Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2: 266–285, 2018.

[8] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[9] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality issues on a gpu in a virtualized environment. In *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*, pages 119–135. Springer, 2014.

[10] S. Breß, S. Kiltz, and Martin Schäler. Forensics on gpu coprocessing in databases -research challenges, first experiments, and countermeasures. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, pages 115–129, 01 2013.

[11] Xavier Bellekens, Greig Paul, James M. Irvine, Christos Tachtatzis, Robert C. Atkinson, Tony Kirkham, and Craig Renfrew. Data remanence and digital forensic investigation for cuda graphics processing units. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 1345–1350, 2015. doi: 10.1109/INM.2015.7140493.

[12] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–25, 2016.

[13] Wenjian HE, Wei Zhang, Sharad Sinha, and Sanjeev Das. igpu leak: An information leakage vulnerability on intel integrated gpu. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 56–61, 2020. doi: 10.1109/ASP-DAC47756.2020.9045745.

[14] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 394–405, 2016. doi: 10.1109/HPCA.2016.7446081.

[15] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243831. URL https://doi.org/10.1145/3243734.3243831.

[16] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*. Citeseer, 2013.

[17] Frederik Dermot Pustelnik, Xhani Marvin Saß, and Jean-Pierre Seifert. Whispering pixels: Exploiting uninitialized register accesses in modern gpus. In *2024 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2024.

[18] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable gpu memory management: Towards recovering raw data from gpu, 2016. URL https://arxiv.org/abs/1605.06610.

[19] Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R. Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W. Fletcher. GPU.zip: On the side-channel implications of hardware-based graphical data compression. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024.

[20] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*, pages 19–33, 2014. doi: 10.1109/SP.2014.9.

[21] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 125–137. IEEE, 2020.

[22] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal {DNN} models with lossless inference accuracy. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[23] Tyler Sorenson and Heidy Khlaaf. LeftoverLocals: Listening to LLM responses through leaked GPU local memory. Blog, January 2024. URL https://blog.trailofbits.com/2024/01/16/leftoverlocals-listening-to-llm-responses-through-leaked-gpu-local-memory/.

[24] Tapadhir Das, Raj Mani Shukla, Mohammed Ben-Idris, and Shamik Sengupta. Vulnerabilities of machine learning algorithms to adversarial attacks for cyber-physical power systems. In Ali Parizad, Hamid Reza Baghaee, and Saifur Rahman, editors, *Smart Cyber-Physical Power Systems: Challenges and Solutions*. Wiley-IEEE Press, 2024.

[25] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with gpu memory exploitation. *Computers & Security*, 102:102115, 2021.

[26] Craig "Torbull" Levine. Bitcoin fiasco. https://play.esea.net/news/12692, 2013.

[27] Dmitry Tanana. Behavior-based detection of gpu cryptojacking, 2024. URL https://arxiv.org/abs/2408.14554.

[28] Aritra Dhar, Clément Thorens, Lara Magdalena Lazier, and Lukas Cavigelli. Ascend-cc: Confidential computing on heterogeneous npu for emerging generative ai workloads. *arXiv preprint arXiv:2407.11888*, 2024.

[29] NVIDIA Corporation. Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2024.

[30] NVIDIA Corporation. Nvidia confidential computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/, 2024.

[31] Michael Larabel. Amd "trusted memory zone" encrypted vram support coming to their linux gpu driver. https://www.phoronix.com/news/AMD-Trusted-Memory-Zone, 2019.

[32] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. G-safe: Safe gpu sharing in multi-tenant environments. *arXiv preprint arXiv:2401.09290*, 2024.

[33] NVIDIA Corporation. Cuda c++ programming interface. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-interface, 2024.

[34] Advanced Micro Devices Inc. C++ language extenstions - hip documentation. https://www.amd.com/en/products/software/rocm.html, 2024.

[35] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.

[36] Bang Di, Jianhua Sun, and Hao Chen. A study of overflow vulnerabilities on gpus. In *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings 13*, pages 103–115. Springer, 2016.

[37] Andrea Miele. Buffer overflow vulnerabilities in cuda: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12:113–120, 2016.

[38] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan.

[39] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security and Privacy*, 10(6):84–87, November 2012. ISSN 1540-7993. doi: 10.1109/MSP.2012.152.

[40] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in Turing-Complete Return-Oriented programming. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, August 2012. USENIX Association. URL https://www.usenix.org/conference/woot12/workshop-program/presentation/Homescu.

[41] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[42] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937032. doi: 10.1145/1315245.1315313. URL https://doi.org/10.1145/1315245.1315313.

[43] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966919.

[44] Ajin Deepak. ROP chains on ARM64, 2023. URL https://infosecwriteups.com/rop-chains-on-arm64-6ff10368798f.

[45] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, 2008.

[46] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing T4 GPU via microbenchmarking. *CoRR*, abs/1903.07486, 2019. URL http://arxiv.org/abs/1903.07486.

[47] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. Decoding cuda binary. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 229–241, 2019. doi: 10.1109/CGO.2019.8661186.

[48] PaX Team. Address space layout randomization (aslr). https://pax.grsecurity.net/docs/aslr.txt, 2001.

[49] NVIDIA Corporation. NVIDIA CUDA Compiler Driver, 2024. URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[50] Microsoft Corporation. Data execution prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2003.

[51] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. Gmod: a dynamic gpu memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359863. doi: 10.1145/3243176.3243194. URL https://doi.org/10.1145/3243176.3243194.

[52] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. Efficient buffer overflow detection on gpu. *IEEE Trans. Parallel Distrib. Syst.*, 32(5):1161–1177, May 2021. ISSN 1045-9219. doi: 10.1109/TPDS.2020.3042965. URL https://doi.org/10.1109/TPDS.2020.3042965.

[53] Jiang Wu, Bang Di, Jianhua Sun, Hao Chen, Xionghu Zhong, DaoKun Hu, and Chenlin Huang. A fast and secure gpu memory allocator. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 146–153, 2019. doi: 10.1109/HPCC/SmartCity/DSS.2019.00035.

[54] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 27–41, 2022.

[55] Chaochao Zhang and Rui Hou. Lak: A low-overhead lock-and-key based schema for gpu memory safety. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 705–713, 2022. doi: 10.1109/ICCD56317.2022.00108.

[56] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589102. URL https://doi.org/10.1145/3579371.3589102.

[57] NVIDIA Corporation. Compute Sanitizer — compute-sanitizer 12.6 documentation, 2022. URL https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.

[58] NVIDIA Corporation. CUDA-MEMCHECK, 2013. URL https://developer.nvidia.com/cuda-memcheck.

[59] James Price and Simon McIntosh-Smith. Oclgrind: an extensible opencl device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334846. doi: 10.1145/2791321.2791333. URL https://doi.org/10.1145/2791321.2791333.

[60] Thomas M. Baumann and José Gracia. Cudagrind: Memory-usage checking for cuda. In Andreas Knüpfer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2013*, pages 67–78, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08144-1.

[61] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for gpgpus. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–73, 2017. doi: 10.1109/CGO.2017.7863729.

[62] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI):124–147, 2023.

[63] Khronos Group. Webcl-validator. https://github.com/KhronosGroup/webcl-validator, 2014.

[64] NVIDIA Corporation. Cuda binary utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, 2024.

[65] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, 2023.

[66] NVIDIA Corporation. Pascal mmu format changes. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf, 2016.

[67] Nanyu Zhong, Yueqi Chen, Yanyan Zou, Xinyu Xing, Jinwei Dong, Bingcheng Xian, Jiaxu Zhao, Menghao Li, Binghong Liu, and Wei Huo. Tgrop: Top gun of return-oriented programming automation. In Joaquin Garcia-Alfaro, Rafał Kozik, Michał Choraś, and Sokratis Katsikas, editors, *Computer Security – ESORICS 2024*, pages 130–152, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-70896-1.

[68] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 969–986, 2016. doi: 10.1109/SP.2016.62.

[69] Jannik Pewny, Philipp Koppe, and Thorsten Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 111–126, 2019. doi: 10.1109/EuroSP.2019.00018.

[70] NVIDIA Corporation. Nvidia cublas. https://docs.nvidia.com/cuda/cublas/index.html, 2024.

[71] NVIDIA Corporation. Nvidia cudnn. https://developer.nvidia.com/cudnn, 2024.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[73] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[74] Aravind Prakash and Heng Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 111–120, 2015.

[75] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[76] Cristina Cismaru, Ruxandra Chiroiu Trandafir, and Emil-Ioan Slusanschi. A study of gpu memory vulnerabilities. In *2023 22nd RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–9, 2023. doi: 10.1109/RoEduNet60162.2023.10274918.

[77] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301336. doi: 10.1145/1920261.1920269. URL https://doi.org/10.1145/1920261.1920269.

[78] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009. doi: 10.1109/SP.2009.25.

[79] Federico Cassano, Charles Bershatsky, Jacob Ginesin, and Sasha Bashenko. Safellvm: Llvm without the rop gadgets!, 2023. URL https://arxiv.org/abs/2305.06092.

[80] Nicolas Devillard. Better Security at the Flick of a (Compiler) Switch: Enabling Pointer Authentication and Branch Target Identification. https://newsroom.arm.com/blog/pac-bti, October 2023.

[81] LLVM Developers. Safestack. https://clang.llvm.org/docs/SafeStack.html, 2021.

[82] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[83] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 40–51, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305648. doi: 10.1145/1966913.1966920. URL https://doi.org/10.1145/1966913.1966920.

[84] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K Iyer. Architecture support for defending against buffer overflow attacks. *Coordinated Science Laboratory Report no. UILU-ENG-02-2205, CRHC-02-05*, 2002.

[85] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 555–566, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332453. doi: 10.1145/2714576.2714635. URL https://doi.org/10.1145/2714576.2714635.

[86] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing*

Systems, pages 409–417, 2001. doi: 10.1109/ICDSC.2001.918971.

[87] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, USA, 2014. USENIX Association. ISBN 9781931971164.

[88] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[89] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[90] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595932267. doi: 10.1145/1102120.1102165.

[91] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013. doi: 10.1109/SP.2013.44.

[92] Intel Corporation. A technical look at intel's control-flow enforcement technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.

[93] Alan Mujumdar. Armv8.1-m pointer authentication and branch target identification extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension, 2021.

[94] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017. ISSN 0360-0300. doi: 10.1145/3054924. URL https://doi.org/10.1145/3054924.

[95] Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz. Towards automating code-reuse attacks using synthesized gadget chains. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security – ESORICS 2021*, pages 218–239, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88418-5.

[96] Edward J. Schwartz, Cory F. Cohen, Jeffrey S. Gennari, and Stephanie M. Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1789–1801, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417234. URL https://doi.org/10.1145/3372297.3417234.

[97] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated return-oriented programming attacks on risc-v and arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 30–42, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397049. doi: 10.1145/3545948.3545997. URL https://doi.org/10.1145/3545948.3545997.

Received 7 February 2025