ROLoad: Securing Sensitive Operations with Pointee Integrity

Wende Tan, Yuan Li, Chao Zhang✉, Xingman Chen, Songtao Yang, Ying Liu, Jianping Wu
Tsinghua University, Beijing, China
twd2.me@gmail.com, chaoz@tsinghua.edu.cn

***Abstract*—Sensitive operations (e.g. control-flow transfers) are attractive targets for attackers. To protect them from being hijacked, we propose a new solution ROLoad to guarantee the integrity of their operands, which are loaded from (potentially corrupted) memory. We extend the RISC-V instruction set, implement an FPGA-based prototype of ROLoad, and then demonstrate two specific defense applications. Results show that this solution only costs few extra hardware resources ($< 3.32\%$). However, it could enable many lightweight (e.g. with overheads less than 0.31%) defenses, and provide broader and stronger security guarantees than existing hardware solutions, e.g. ARM BTI and Intel CET.**

I. Introduction

Attackers often utilize memory corruption vulnerabilities to launch attacks, by corrupting operands of sensitive operations, e.g. targets of indirect control-flow transfers, arguments of sensitive API invocations, and allowlists used in security checks. For instance, attackers can corrupt function pointers used by indirect calls to hijack control flows and execute malicious code [1].

Many hardware features have been proposed to facilitate mitigations of such threats. For instance, Intel MPK [2], ARM DACR [3], IMIX [4], and HDFI [5] could *isolate* sensitive data from being corrupted. Moreover, Intel MPX [6] and ARM MTE [7] prevent memory corruptions *at sources* (i.e. memory read and write operations), while ARM PA [8] detects corruptions before data is used *at sinks* (i.e. sensitive operations). Further, Intel CET [9] and ARM BTI [10] check the validity of operands of indirect transfers and provide CFI (control-flow integrity) guarantees. However, most of these features are complex to implement or have high runtime overheads, and are hard to deploy on low-end devices (Section VI). Therefore, an effective and lightweight protection is highly desirable.

In this paper, we propose a hardware-software co-design solution ROLoad, consisting of a new set of hardware instructions, kernel support, and compiler extension, to address these challenges and protect the integrity of sensitive operands. ROLoad provides a weak form of data-flow integrity, namely *pointee integrity*, which ensures pointees are not corrupted (i.e. loaded from read-only memory tagged with specific keys or types). Given this security guarantee, attackers can only feed existing read-only (i.e. non-corrupted) data of expected type to sensitive operations, and thus can hardly launch attacks.

We have built a prototype computer system supporting ROLoad on a field-programmable gate array (FPGA). Specifically, we extended the RISC-V instruction set architecture (ISA) and augmented the processor core in our prototype, a RISC-V Rocket Core, to support the ROLoad-family instructions. We also slightly modified the Linux kernel to set keys of memory pages and handle a new type of page fault. Further, we extended the compiler infrastructure LLVM [11] to provide interfaces for programs to utilize this feature. Finally, we demonstrated two specific applications of ROLoad, i.e. virtual function call protection and type-based forward-edge CFI, to show the effectiveness of ROLoad. Please note that ROLoad is not limited to these two defense applications. *We believe any defense solutions adopting allowlist checks can utilize this feature.*

This work was supported in part by the National Natural Science Foundation of China under Grant 61972224 and U1736209.
978-1-6654-3274-0/21/$31.00 ©2021 IEEE

We evaluated the performance of computer systems supporting ROLoad. Results showed that ROLoad takes few extra hardware resources ($< 3.32\%$). We also evaluated performances of the two defense applications, i.e. virtual call protection and forward-edge CFI. Their average runtime overheads are negligible, i.e. less than 0.31% and 0.09% respectively. But they provide strong security guarantees comparable to software-based counterparts (e.g. the one in the Clang compiler [12]) and even heavier hardware features (e.g. ARM PA [13]). Specifically, it enables broader and stronger defenses than ARM BTI [10] and Intel CET [9] for forward-edge transfers. This shows that ROLoad is practical and suitable for devices with limited resources (e.g. IoT devices).

In summary, we make the following contributions:

- We propose a novel lightweight solution ROLoad to secure sensitive operations by enforcing *pointee integrity*.
- We introduce a new set of instructions which only load data from read-only memory pages tagged with specific keys to provide pointee integrity, and augment a RISC-V Rocket Core to support these instructions on the RISC-V ISA.
- We build a prototype system supporting this feature on an FPGA, and provide interfaces for software to use this feature.
- We demonstrate two specific defense applications of this feature, illustrating its usefulness in practice.
- We conduct a thorough evaluation of the proposed solution, showing ROLoad is lightweight, effective, and practical.

II. Design

A. The Challenge

Few hardware-based mitigations have been deployed by manufacturers in real world, though many have been proposed, due to the compatibility and overhead concerns. For those deployed ones, e.g. ARM PA [8] and ARM MTE [7], many still require non-negligible hardware resources and/or memory or runtime overheads. Therefore, to deploy strong protections on systems with limited resources (e.g. IoT devices), alternative mitigations which are lightweight, compatible, and low hardware-cost are highly desirable.

B. Threat Model

The assumptions made in our threat model are consistent with prior work in computer security [14], [4], [5] and are outlined as follows.

- We assume that one or more memory-corruption vulnerabilities exist in victim programs, allowing adversaries to repeatedly read from or write to arbitrary readable/writable addresses.
- We assume that DEP [15] is deployed and code is immutable.
- We assume that the operating system kernel is trusted and has no vulnerabilities or cannot be exploited by adversaries.
- As a hardware-software co-design approach, we assume that the hardware is trusted and cannot be tampered.

C. Intuition

Most sensitive operations essentially only allow a list of immutable legitimate operand values. Examples of such allowlists include C++ virtual function tables, customized function pointer tables, format strings, and hardcoded configurations. On one hand, we can place these immutable values in different types of *tamper-proof areas* to protect them from corruption. On the other hand, we can block tainted

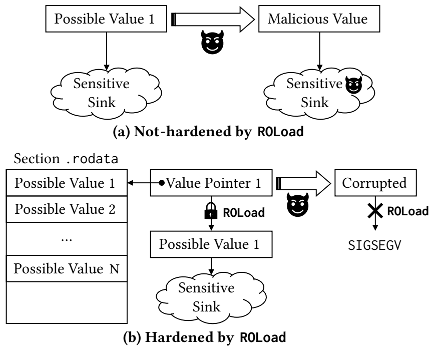

Fig. 1: Intuition of ROLoad's Design

(or corrupted) data from being used at sensitive operations, i.e. only data from specific type of tamper-proof areas are allowed to be used at sensitive operations, to mitigate memory-corruption attacks.

In this way, only values in the specific type of allowlists can be used at sensitive operations (i.e. sinks), rendering attacks extremely hard. Figure 1 demonstrates this intuition.

D. Design Choices

On real modern computer systems with paging (e.g. RISC-V, ARM, MIPS, x86, etc.) or with physical memory protection (e.g. RISC-V PMP [16], ARM Memory Protection Unit [17]), intuitively, we can choose read-only memory pages (regions) as the tamper-proof areas to place allowlists. Different types of allowlists may be placed in different read-only pages which are tagged with different *keys* to differentiate from each other.

At runtime, we ensure a sensitive operation can only be fed with data loaded from a read-only memory page tagged with a specific key, which is non-corrupted and of the expected type, rendering attacks extremely hard. To do so, we introduce a new set of simple and lightweight *load* instructions, ROLoad-family instructions. Unlike existing load instructions, ROLoad-family instructions can only access read-only pages with specific keys (tags), otherwise trigger page faults. In other words, it can guarantee the pointee integrity, i.e. data loaded by these instructions are not corrupted, and can secure operands of sensitive operations.

The ROLoad design is analogous and complementary to the widely deployed DEP defense (NX-bit). DEP ensures only pages without writable permissions can be executed, while ROLoad ensures only pages without writable permissions can be loaded and used as sensitive operands. DEP focuses on code, but ROLoad focuses on data. As we will discuss later, the ROLoad-family instructions can be easily implemented, have practical security applications, and introduce very low runtime overheads, similar to DEP. We believe that both of them are necessary in a secure computer system design.

In the rest of this paper, we mainly focus on systems with a paging mechanism, i.e. a memory management unit (MMU). However, ROLoad is not limited to these systems. It is even easier to implement ROLoad on systems only with physical memory protection mechanisms (e.g. embedded systems), making it applicable to a wide range of systems, including low-end IoT devices.

E. The ROLoad Solution

Figure 2 shows the overview of ROLoad, which consists of the following three major components:

- the processor core (e.g. a RISC-V Rocket Core) at the bottom, responsible for supporting the ROLoad-family instructions.
- the operating system kernel (e.g. the Linux kernel) in the middle, responsible for configuring the keys of memory pages and handling exceptions caused by permission check or key check failures.

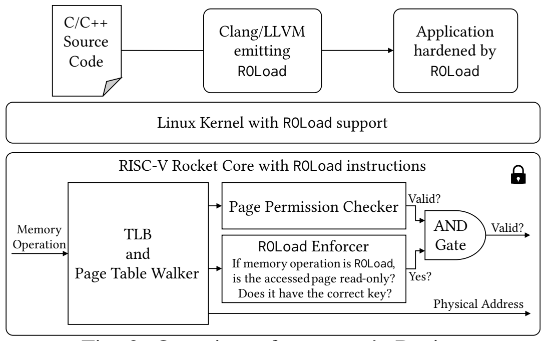
C/C++ Source Code
Clang/LLVM emitting ROLoad
Application hardened by ROLoad
Linux Kernel with ROLoad support
RISC-V Rocket Core with ROLoad instructions
Memory Operation
TLB and Page Table Walker
Page Permission Checker
Valid?
ROLoad Enforcer
If memory operation is ROLoad, is the accessed page read-only? Does it have the correct key?
Yes?
AND Gate
Valid?
Physical Address

Fig. 2: Overview of ROLoad's Design

- the compiler infrastructure (e.g. Clang/LLVM) at the top, for hardening applications with the proposed ROLoad-family instructions.

1) ROLoad-family instruction support: ROLoad-family instructions only load data from read-only memory pages with specific keys. This permission check and key check are enforced by the hardware processor core efficiently, providing strong security guarantees with low runtime overheads.

First, we augment the MMU inside a processor to support a newly introduced field, namely *key*, in each page table entry. In this way, each page is associated with a key, and different pages may share a same key. These keys can be used to represent the kinds, types, or classes of the data stored in these pages, and the actual meanings of the keys are defined by security applications.

Then, ROLoad-family instructions will use these keys to ensure that the accessed page is the right one by comparing the keys specified in these instructions' operands with the keys of the accessed pages. Specifically, we add a light extra logic into the MMU to check whether the accessed page is read-only and whether its key is the same as the key of the requesting instruction when ROLoad-family instructions are executed. If yes, then the ROLoad-family instructions behave in the same way as normal load instructions. Otherwise, the processor generates a page fault. The output of this logic is then ANDed with the original output of the page permission control logic. Thus, the conventional page permission check and the newly introduced ROLoad checks are done in parallel.

2) Kernel Support: The kernel is responsible for configuring keys for read-only pages and handling page faults in case of attacks.

a) Page key setting up: Before a process gets started, the kernel helps the process set up its page keys, either by itself during executable loading, or by providing APIs for user-mode processes.

b) Page fault handling: If the read-only permission check or the page key check fails, a page fault will be generated by the processor core, and the kernel is thus responsible for handling this page fault to stop the running process. Specifically, the kernel will first differentiate this special type of page faults from regular page faults, and then halt the faulting process, which is being attacked.

It is worth noting that, this solution is applicable to not only user-mode programs, but also operating system kernels, as well as systems without kernel-user separation.

3) Compiler Support: We extend the compiler to provide support for applications that want to utilize the ROLoad-family instructions.

The compiler first determines *where sensitive operations are* and *what their allowlists are*, and then places these allowlists into different groups of read-only pages (e.g. sections in executable files) with different keys (i.e. different tamper-proof areas).

Then, it slightly modifies target programs by replacing regular memory load instructions at each sensitive operation with ROLoad-family instructions with special keys as operands.

TABLE I: Lines of code of each ROLoad component.

Components	Language	#Lines of Code Added	Modified	Total
RISC-V Processor	Chisel	29	30	59
Linux Kernel	C	118	3	121
LLVM Back-end	C++ and TableGen	268	2	270
Total	–	415	35	450

III. Implementation

ROLoad can be applied to any system with paging or physical memory protection support. For simplicity, we choose to extend the RISC-V ISA to support ROLoad-family instructions, and implement a prototype based on an FPGA, which consists of three components with hundreds of lines of code modification as listed in Table I.

A. RISC-V Processor

We chose to extend the RISC-V ISA. The design of ROLoad-family instructions is consistent with RISC-V's design philosophy, e.g. simple, easy to use, and with low cost [18]. It can be integrated into RISC-V ISA with only slight changes.

Specifically, we extend the mnemonics of regular memory load operations (e.g. `ld`) to corresponding ROLoad versions (e.g. `ld.ro`). To optimize the program size, we also extend the C extension of RISC-V ISA and implemented the compressed encoding version of `ld.ro`, namely `c.ld.ro`, i.e. the ROLoad version of `c.ld`.

After picking optimal instruction encodings, we augment the classes `Instructions`, `IDecode`, and `RVCDecoder` in the Rocket Chip Generator to generate decoders of the newly introduced `ld.ro`-family instructions and `c.ld.ro`. Once these instructions are decoded, they will issue a new type of memory operations carrying the keys decoded from the instructions. We also add this new type to `MemoryOpConstants`, which will later be used by Class `TLB` to perform the page permission and page key checks.

We locate the code in Class `TLB` responsible for permission checks, and add the extra read-only permission check logic explained in Section II-E. We also add the newly introduced *key* field described in Section II-E to each page table entry and each TLB entry. Page table entries are fixed-size of 64 bits on 64-bit RISC-V systems, and we reuse the previously reserved top 10 bits of each page table entry. Our modifications are consistent with others.

B. Linux Kernel

Before a process gets started, the kernel helps the process set up its page keys. To do so, we add functions into `arch/riscv` in the Linux kernel to handle page keys at each level of MMU abstraction so that user-mode processes can finally use `mmap()` and `mprotect()` system calls to set up page keys for themselves.

After a process starts executing, once the processor raises a page fault, the kernel is responsible for handling it. We slightly augment the page fault handling procedure of the RISC-V ISA in the Linux kernel (`arch/riscv/mm/fault.c`). It first distinguishes load page faults raised by ROLoad-family instructions from benign load page faults raised by regular load instructions. If the load page faults are raised by ROLoad-family instructions because of read-only permission check failure or key check failure, the modified Linux kernel will send a segmentation fault (`SIGSEGV`) signal to the faulting process to warn and/or kill it.

C. LLVM Compiler Back-end

A compiler back-end emits low-level machine instructions for programs written in high-level languages. We extend the RISC-V back-end of LLVM compiler infrastructure to translate ROLoad-family instructions to machine code and provide interfaces for applications to utilize via LLVM IR instructions.

Following the design pattern of LLVM, we add the descriptions of `ld.ro`-family instructions written in TableGen language [19] into the *td files* of RISC-V target. This allows the assembler to recognize `ld.ro`-family instructions and generate correct machine code.

Metadata Interface: The interfaces are a new type of metadata, namely `ROLoad-md` metadata. Users (e.g. defense solutions) associate LLVM IR load instructions of interest with this metadata to indicate that this IR load instruction needs to be further protected by a ROLoad-family instruction. Keys that will be encoded into ROLoad-family instructions are stored in the `ROLoad-md` metadata as well. We also associate this type of metadata with each `MachineMem Operand` when the compiler is generating machine instructions.

Instruction Emission: To emit `ld.ro`-family instructions, we write a machine code pass to visit all LLVM machine instructions and replace all `ld`-family instructions whose machine memory operands have `ROLoad-md` metadata, with `ld.ro`-family instructions. Since `ld.ro`-family instructions no longer have any address offset encoded in their immediates, extra `addi` instructions may also be inserted.

IV. Applications

Programs can utilize the ROLoad-family instructions to protect the integrity of sensitive operands. In this section, we demonstrate two specific defense applications of ROLoad, which protect virtual function calls and general forward-edge control-flow transfers respectively, and then discuss other potential applications.

A. Virtual Function Call Protection

Virtual function calls (vcalls) are the most common type of indirect calls used in large software products (e.g. browsers) written in C++ [20]. In practice, a common type of attacks against these software products is the VTable hijacking attack, which leverages memory corruption vulnerabilities to tamper with virtual function table pointers (vptrs) associated with objects and then hijacks the control flow of vcalls.

ROLoad can be utilized to mitigate this threat and can provide a stronger security guarantee than VTint [21], which enforces that virtual function pointers used at vcalls are loaded from read-only memory and stops VTable corruption and injection attacks. Unlike the software-based solution VTint, our hardware-software co-design solution ROLoad will introduce negligible performance overheads.

Specifically, we first classify VTables based on class types and move them into read-only pages with corresponding keys. Then, we can replace VTable loading instructions with ROLoad-family load instructions, to enforce that virtual function pointers are read from read-only memory pages with matching keys and stop most VTable hijacking attacks. Since VTables have already been computed and stored into read-only pages by modern compilers, our movement of VTables and replacement of load instructions will work fine without modifying the content of VTables. We can infer that the runtime performance overhead will be negligible.

B. Type-Based Forward-Edge CFI

In addition to virtual function calls, programs also have many general indirect calls, especially the programs with callback functions or dynamic dispatching. Targets of these indirect control-flow transfers (ICT) are sensitive operands, too. In practice, adversaries can exploit vulnerabilities to corrupt these ICT targets and hijack control flows. CFI solutions [14] restrict ICT targets to a limited set at runtime.

There are several types of ICTs, including returns, indirect calls and jumps. In our prototype, we only focus on the most common forward-edge control-flow transfers, i.e. indirect calls, and aim at providing a type-based CFI policy. Under this policy, indirect calls can only jump to address-taken functions which have matching function types.

```
typedef void (*func1_t)(...);
typedef int (*func2_t)(...);
func1_t func1;
func2_t func2;
// ...
func1 = foo;
func2 = bar;
// ...
func1();
func2();
```

Listing 1: Pseudo code of two indirect call examples.

```
-lui  a0, 0x11
-addi a0, a0, 604    # foo
+lui  a0, 0x67       # mem page addr
+addi a0, a0, 8      # gfpt_foo
 sd   a0, -1608(gp)  # func1
 ...
-lui  a0, 0x11
-addi a0, a0, 616    # bar
+lui  a0, 0x68       # gfpt_bar
 sd   a0, -1600(gp)  # func2
```

Listing 2: Assembly code of initializing 2 function pointers for ROLoad.

```
 1   ld    a0, -1608(gp) # func1
 2  +ld.ro a0, (a0), 111
 3   jalr  a0
 4   ld    a0, -1600(gp) # func2
 5  +ld.ro a0, (a0), 222
 6   jalr  a0
 7  +.section .rodata.key.111
 8  +gfpt_foo: .quad foo
 9  +.section .rodata.key.222
10  +gfpt_bar: .quad bar
```

Listing 3: Assembly code of two indirect calls, hardened by ROLoad.

Therefore, we first identify all address-taken functions and their types, and place their addresses into read-only memory pages with keys which are equivalent to function types. Then, we modify indirect call instructions to only use function addresses loaded from these read-only memory pages with matching keys. In this way, ROLoad ensures that indirect calls only transfer to address-taken functions with correct types, providing a type-based forward-edge CFI.

Listing 1, 2, and 3 demonstrate an example of this application. In this example, two global function pointer tables (GFPTs) are created and put in read-only memory pages with keys of 111 and 222, and are both initialized to an array of addresses of address-taken functions, i.e. foo and bar (line 7∼10 in Listing 3). Original function pointers are replaced with pointers to GFPT items, as shown in Listing 2. When function pointers are used, the corresponding items are loaded by ld.ro instructions (line 2 and 5 in Listing 3), ensuring that the function addresses are loaded from the read-only arrays (i.e. GFPTs) with the correct keys of 111 and 222. Then, these loaded items, which are the actual addresses of the callee functions, are used by the following indirect call instructions (line 3 and 6 in Listing 3).

C. Other Application Scenarios

Aforementioned two applications share a similar principle: targets allowed in sensitive operations reside in specific allowlists, i.e. sets of legitimate functions. We believe that *all allowlist-based defenses* can be enhanced by ROLoad to achieve better performance and security.

Specifically, given an allowlist check, we could first place allowlists into read-only memory pages tagged with unique keys, and then transform the allowlist check to a ROLoad check, i.e. ensuring the targets are in allowlists (i.e. loaded from read-only pages tagged with the correct keys). It thus stops corrupted values from being used in sensitive operations with low overheads.

For instance, it can be applied to backward control-flow transfers (i.e. return instructions) too, where the allowlists are sets of legitimate return sites. It can also be applied to the Linux kernel in which many structure pointers (e.g. device descriptors, operation structures) have limited value choices, where the sets of available device descriptors or operation structures can be viewed as the allowlists.

V. Evaluation

In this section, we evaluate our prototype computer system featuring ROLoad and programs hardened by ROLoad-family instructions to answer the following questions:

- **RQ1: Lightweight Hardware Implementation:** Is it low-cost and lightweight to implement ROLoad-family instructions on hardware (e.g. on FPGAs)? (Section V-A)
- **RQ2: Low Runtime Overheads:** Does a system with ROLoad run as fast as an unmodified system? (Section V-B) How much execution time and memory overhead do programs hardened by ROLoad incur? (Section V-C1)
- **RQ3: Strong Security Guarantees:** What security guarantees do ROLoad and programs hardened by ROLoad provide? (Section V-C2)

TABLE II: Configuration of our prototype computer system.

Components	Configurations
ISA Extensions	RV64IMAC with M, S, and U modes
Caches	32KiB 8-way L1I$, 32KiB 8-way L1D$
TLBs	32-entry I-TLB, 32-entry D-TLB (default)
Peripherals	Xilinx MIG for a 4GiB DDR3 SO-DIMM Xilinx AXI Ethernet Subsystem, 64KiB boot ROM

A. Hardware Resource Cost

To evaluate the hardware resource cost of ROLoad-family instructions, we instantiate the original and the modified RISC-V Rocket Cores using the Rocket Chip Generator and synthesise them on an FPGA. Table II shows the overall configurations of our prototype computer systems. These systems can boot Linux from a network.

We synthesise and map our prototype computer systems to an FPGA on a commodity Xilinx Kintex 7 (XC7K420T) FPGA board using Xilinx Vivado 2019.1. The target frequency of synthesis is $F_{target} = 125.00$MHz, which is approximately the maximum frequency that RISC-V Rocket Cores on our FPGA can operate at. To be fair, we also synthesise the RISC-V Rocket Cores out of context without peripheral to evaluate the hardware resource cost of RISC-V Rocket Cores themselves. Results are presented in Table III.

The results show that the extra hardware resource cost on an FPGA measured in terms of the number of LUTs and FFs are both low ($< 3.32\%$). Besides, the maximum frequency is approximately not affected. This gives us evidence that ***ROLoad-family instructions can be adopted in real-world systems with low hardware cost.***

B. Performance Evaluation

We evaluate the overall performance of computer systems supporting ROLoad using the FPGA-based prototype system synthesised in the previous subsection.

The operating system kernel is Linux kernel 5.4.11 with the official RISC-V support, and is compiled by GCC 9.2.0 for RV64IMAC. All executables and input data of benchmarks are fetched from a network file system and stored into the memory before execution.

We used an unmodified version of SPEC CINT2006 benchmarks [22]. Among them, *400.perlbench* is excluded because of compilation failure. These benchmarks are compiled and instrumented by Clang and LLVM 11.0.0 with -O2 -s -static, and linked with musl libc 1.2.0. The musl libc is compiled and instrumented by the same compiler suite. The -z separate-code flag of the linker is also turned on to separate the read-only data from code sections, otherwise the linker will store read-only data into the pages that are both readable and executable, violating the read-only requirement of ROLoad-family instructions.

We selected the standard *reference* workloads. All benchmarks are executed on three systems, i.e. the baseline system (the system without ROLoad support), the processor-modified system, and the processor-and-kernel-modified system, respectively. Execution time and memory usages are both measured, in terms of the number of

TABLE III: Hardware resource cost of systems without and with `ROLoad` when synthesised on an FPGA.

	RISC-V Rocket Cores				Whole Systems					
	#LUT	%	#FF	%	#LUT	%	#FF	%	Worst Setup Slack (ns)	F_{max} (MHz)
without `ld.ro`	20,722	–	11,855	–	37,428	–	29,913	–	0.119	126.89
with `ld.ro`	21,021	+1.44291	12,248	+3.31506	37,765	+0.90040	30,347	+1.45087	0.099	126.57

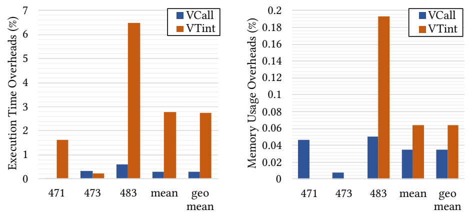

Fig. 3: Relative runtime and memory overheads of VCall and its competitor VTint, based on SPEC CINT2006 benchmarks.

clock cycles and KiB respectively. Due to the limitation of computing power, each experiment needs ∼6 *days* to finish.

Since the benchmarks can finish successfully, we can infer that both processor and kernel modifications do not break the backward compatibility. The results show that, both modifications introduce runtime and memory overheads ∼ 0%, which means ***a system with*** **`ROLoad`** ***can run as fast as an unmodified system***.

C. Defense Application Evaluation

`ROLoad` can be utilized to secure sensitive operations and harden programs. In this section, we evaluate its runtime performance and security guarantees based on two specific applications.

1) Performance: To evaluate the performance impact of two security applications in Section IV, we measured the execution time and memory usage of the benchmarks hardened by these applications. To compare, we also measured the benchmarks hardened by their competitors. Other experimental setups are the same as the setups presented in Section V-B. In addition, the baseline system in this section is the system with both processor and kernel modifications.

a) Virtual Call Protection Performance: For the application of protecting virtual function calls in Section IV-A (*VCall* for short), we measured the original 3 C++ benchmarks in SPEC CINT2006 and the version hardened by VCall. To compare, we also measured the benchmarks hardened by VTint [21]. We ported VTint to the RISC-V platform, and utilized range-based checks before VTable loading to check whether VTables are loaded from read-only memory.

The experimental results are shown in Figure 3. The average relative execution time overhead of VCall is very low (0.303%), and is better than that of VTint (2.750%). The memory overhead of VCall is 0.0347%, and that of VTint is 0.0644%. Both are negligible, but the instrumentation of VTint has enlarged the code section, thus introduces slightly higher memory overheads.

b) Forward-edge CFI Performance: For the type-based forward-edge CFI in Section IV-B (*ICall* for short), we measured the original SPEC CINT2006 benchmarks and the version hardened by ICall. To compare, we also measured the benchmarks hardened by the original CFI. We ported the CFI implementation to RISC-V, by inserting an ID (which is equivalent to `nop` at the ISA level) at the beginning of each function, and adding checks before indirect calls to check whether the indirect call targets have the correct ID.

The experimental results are shown in Figure 4 and Figure 5. The average relative execution time overhead of ICall is almost zero, and is better than that of CFI (9.073%). Please note that our ICall has lower execution time overheads than our VCall, because ICall uses a unified key for all VTables and uses other keys for other function pointers, and thus has better TLB and cache locality. The memory overheads of ICall is 0.0859%, and that of CFI is 0.0500%. In the

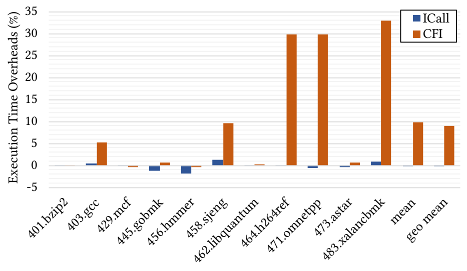

Fig. 4: Relative runtime overheads of ICall and its competitor CFI, based on SPEC CINT2006 benchmarks.

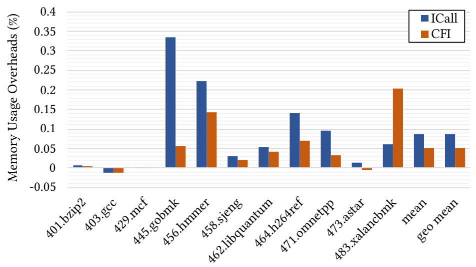

Fig. 5: Relative memory overheads of ICall and its competitor CFI, based on SPEC CINT2006 benchmarks.

ICall application, we store extra function pointers into pages with different keys, thus introduce slightly higher memory overheads.

We can conclude that ***defense solutions that utilize*** **`ROLoad`** ***can protect real-world software with very little extra runtime cost***.

2) Security: The security guarantees provided by `ROLoad` depend on both the underlying hardware primitives and the defense applications. The security of the primitive, i.e. `ROLoad`-family instructions, will be compared with other solutions later in Section VI.

The VCall application first implements the same functionality as VTint, i.e. limiting VTables are loaded from read-only memory, and then it uses keys to differentiate between VTables of different types. Thus, the security guarantees of the VCall application are at least stronger than those of VTint but with lower runtime overheads.

The ICall application restricts the targets of indirect calls to those with the matching types (i.e. keys), and thus implements a type-based CFI. It provides security guarantees stronger than coarse-grained CFI solutions (e.g. [23], [24]), which do not consider function types.

Both VTable protection and type-based CFI have been extensively studied and their advantages and weaknesses are well-known [21], [12], [13]. We can confirm that ***our*** **`ROLoad`** ***solution provides comparable security guarantees but with much lower runtime performance overheads***.

D. Remarks

It is important to note that like prior lightweight hardware-based solutions, e.g. DEP [15], ARM BTI [10], and Intel CET [9], our `ROLoad` solution could also suffer from pointee reuse attacks as pointees in read-only pages with keys could be reused by adversaries [1]. For example, a sophisticated adversary can corrupt pointers to reuse existing data in any read-only memory pages with matching keys, to bypass `ROLoad`-based checks. However, the remaining attack surface is minimal, as attackers can only feed values in the specific allowlists to sensitive operations.

VI. Related Work

In this section, we review some related defense mechanisms and compare ROLoad with them.

Control-Flow Integrity. CFI is a popular defense against control-flow hijacking attacks [14], [12], which in general restricts ICTs only jump to specific allowlists at runtime. Many software-based CFI solutions have been proposed to reduce the runtime overheads as much as possible. But, the overheads are still rather high for low-end systems (e.g. IoT devices). Compared with CFI mechanisms with the same security strength (i.e. fine-grained type-based CFI), the defense scheme ICall (Section IV-B) based on ROLoad introduces negligible overheads. Thus, ROLoad is more practical on systems with limited resources.

Several recent hardware features have been proposed to facilitate CFI mechanisms, including Intel CET [9] and ARM BTI [10]. However, Intel CET and ARM BTI require an extra architectural state, which needs to be maintained when the OS kernel is switching context or handling interrupts. Extra states could increase the complexity of OS kernels and processor cores [18]. Besides, the security strength of both of those is much weaker than the type-based CFI provided by ROLoad, since they are only coarse-grained and allow each ICT to jump to a shared large allowlist. ROLoad can provide stronger defense than these features. In addition, it also has broader security applications not limited to CFI.

Data-Flow Integrity (DFI). DFI tries to ensure that the data-flow graph of user-mode programs [25] or OS kernels [26] is not violated at runtime. Data-flows have their sources and sinks, and DFI mechanisms can be classified into three groups:

a) Prevention at sources: Intel MPX [6] performs bound checks and prevents overflow-based memory corruptions, but practices show high runtime overheads [27]. ARM MTE [7] provides an ingenious mechanism of associating words with *tags*, and can be used to build hardware-assisted memory safety solutions [28]. But, it is expected to take considerable hardware resources to implement and have relatively high runtime overheads, and we are unaware of any ARM MTE implementation currently.

b) Data-flow isolation: HDFI [5] can provide strong data-flow isolation and security guarantees by associating several words with one-bit tags. However, because of these tags, it is complex and also requires considerable hardware resources to implement. IMIX [4] is a lightweight page-grained data-flow isolation solution, which associates pages with one-bit tags. Similar to other isolation mechanisms, it provides a coarse-grained isolation. Moreover, appropriate deployments of these mechanisms may require considerable manual efforts, e.g. to clearly specify the boundary of sensitive data.

c) Detection at sinks: Solutions like ARM PA [8] could detect corruptions before sensitive data are used. The hardware feature ARM PA enables defenses (e.g. PARTS [13]) to validate the integrity of pointers at all sinks (i.e. pointer dereferences). However, ARM PA relies on the kernel to protect encryption keys, thus is not suitable for systems without kernel-user separation. Instead, ROLoad also works at sinks, but provides a lightweight and practical pointee integrity mechanism, suitable for a wide range of systems.

VII. Conclusion

In this paper, we have presented a lightweight hardware-software co-design solution ROLoad, which can be utilized by program hardening solutions to protect sensitive operations from being hijacked. We have extended the RISC-V ISA and implemented an FPGA-based prototype, and have demonstrated two specific defense applications with low performance overheads and high security guarantees.

References

[1] F. Schuster *et al.*, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Proc. IEEE S&P'15*, 2015, pp. 745–762.

[2] Intel. (2019) Intel(r) 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. [Online]. Available: https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf

[3] A. Limited. (2018) Arm(r) architecture registers armv8, for armv8-a architecture profile. [Online]. Available: https://static.docs.arm.com/ddi0595/b/DDI_0595_ARM_architecture_registers.pdf

[4] T. Frassetto *et al.*, "IMIX: In-process memory isolation extension," in *Proc. USENIX Security '18*, Aug. 2018, pp. 83–97.

[5] C. Song *et al.*, "Hdfi: Hardware-assisted data-flow isolation," in *Proc. IEEE S&P'16*, 2016, pp. 1–17.

[6] Intel. (2013) Introduction to intel(r) memory protection extensions. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[7] A. Limited. (2019) Memory tagging extension: Enhancing memory safety through architecture. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety

[8] ——. (2017) Arm(r) architecture reference manual armv8, for armv8-a architecture profile. [Online]. Available: https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

[9] Intel. (2019) Control-flow enforcement technology specification. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[10] ARM. (2019) Bti: Branch target identification - arm developer. [Online]. Available: https://developer.arm.com/docs/ddi0602/d/base-instructions-alphabetic-order/bti-branch-target-identification

[11] C. Lattner *et al.*, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proc. IEEE CGO '04*, 2004, pp. 75–86.

[12] C. Tice *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. USENIX Security '14*, Aug. 2014, pp. 941–955.

[13] H. Liljestrand *et al.*, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proc. USENIX Security '19*, pp. 177–194.

[14] M. Abadi *et al.*, "Control-flow integrity," in *Proc. ACM CCS '05*, 2005, pp. 340–353.

[15] Microsoft. (2018) Data execution prevention. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[16] A. Waterman *et al.*, "The risc-v instruction set manual, volume i: User-level isa, document version 20191213," *RISC-V Foundation*, Dec. 2019.

[17] ARM. (2016) Memory protection unit (mpu). [Online]. Available: https://static.docs.arm.com/100699/0100/armv8m_architecture_memory_protection_unit_100699_0100_00_en.pdf

[18] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[19] L. Project. (2020) Tablegen language reference. [Online]. Available: https://llvm.org/docs/TableGen/LangRef.html

[20] N. Burow *et al.*, "Cfixx: Object type integrity for c++ virtual dispatch," in *Proc. NDSS '18*, 2018.

[21] C. Zhang *et al.*, "Vtint: Protecting virtual function tables' integrity," in *Proc. NDSS '15*, 2015.

[22] S. P. E. Corporation. (2006) Spec cint2006 benchmarks. [Online]. Available: https://www.spec.org/cpu2006/CINT2006/

[23] C. Zhang *et al.*, "Practical control flow integrity & randomization for binary executables," in *Proc. IEEE S&P'13*. IEEE, 2013, pp. 559–573.

[24] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. USENIX Security '13*, Aug. 2013, pp. 337–352.

[25] M. Castro *et al.*, "Securing software by enforcing data-flow integrity," in *Proc. USENIX OSDI '06*, 2006, pp. 147–160.

[26] C. Song *et al.*, "Enforcing kernel security invariants with data flow integrity," in *Proc. NDSS '16*, 2016.

[27] O. Oleksenko *et al.*, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, Jun. 2018.

[28] K. Serebryany *et al.*, "Memory tagging and how it improves c/c++ memory safety," *arXiv preprint arXiv:1802.09517*, 2018.