Nyman, Thomas; Dessouky, Ghada; Zeitouni, Shaza; Lehikoinen, Aaro; Paverd, Andrew; Asokan, N.; Sadeghi, Ahmad-Reza

HardScope: Hardening Embedded Systems Against Data-Oriented Attacks

Published in:
Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019

DOI:
10.1145/3316781.3317836

Published: 02/06/2019

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

HardScope: Hardening Embedded Systems Against Data-Oriented Attacks

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Ghada Dessouky
Technische Universität
Darmstadt, Germany

Shaza Zeitouni
Technische Universität
Darmstadt, Germany

{ghada.dessouky,shaza.zeitouni}@trust.tu-darmstadt.de

Aaro Lehikoinen
Aalto University, Finland
aaro.j.lehikoinen@aalto.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

N. Asokan
Aalto University, Finland
asokan@acm.org

Ahmad-Reza Sadeghi
Technische Universität
Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

Abstract

Memory-unsafe programming languages like C and C++ leave many (embedded) systems vulnerable to attacks like control-flow hijacking. However, defenses against control-flow attacks, such as (fine-grained) randomization or control-flow integrity are ineffective against data-oriented attacks and more expressive *Data-oriented Programming* (DOP) attacks that bypass state-of-the-art defenses.

We propose *run-time scope enforcement* (RSE), a novel approach that efficiently mitigates *all currently known DOP attacks* by enforcing compile-time memory safety constraints like variable visibility rules at run-time. We present Hardscope, a proof-of-concept implementation of hardware-assisted RSE for RISC-V, and show it has a low performance overhead of 3.2% for embedded benchmarks.

1 Introduction

Data-oriented attacks can influence program behavior without the need to modify control-flow data. Instead, they corrupt variables used by the program's decision making, or leak sensitive information from program memory. Such attacks are called non-control-data attacks [7]. Non-control-data attacks have been shown to allow attackers to forge user credentials, change security critical configuration parameters, bypass security checks, and escalate privileges. Recent work shows that it is even possible to generalize data-oriented attacks to construct full-blown malicious attacks with Turing-complete expressiveness, called *Data-Oriented Programming* (DOP) [15]. Such attacks are executed by carefully corrupting only non-control data over time to chain together sequences of operations on attacker-controlled input. DOP provides similar capabilities to attackers as return-oriented programming [26], but without breaking the victim program's control-flow integrity. This, combined with the ability for DOP to reuse virtually any data, makes preventing DOP attacks a significant and open challenge.

Existing defenses against control-flow attacks cannot prevent data-oriented attacks. Some defenses against non-control-data attacks (e.g., [5, 24]) protect individual pieces of (security-critical) data. Hu et al. [15] discuss various existing schemes that could reduce the number of DOP attacks, including memory safety, data-flow integrity, fine-grained data-plane randomization, and hardware/software fault isolation. However, they explain that existing approaches are either too coarse grained, or result in prohibitively high performance overheads. Without viable alternatives, and because effective defenses against control-flow attacks are already being deployed, DOP is likely to become the next appealing attack technique for run-time exploitation.

Goals and Contributions. We propose a new *efficient* defense against data-oriented attacks that effectively prevents *all currently known* DOP attacks. It can also be configured to prevent control-flow hijacking. The intuition behind our approach is simple: In *block structured languages* every variable has a *lexical scope*, denoting the block(s) of source code in which the variable is visible. All correct compilers enforce *variable scope* at compile-time by checking these variable visibility rules. All currently known DOP attacks, and many data-oriented attacks in general, violate variable scope rules at run-time, since there is no equivalent enforcement. Consequently, mechanisms for variable scope enforcement *at run-time* can significantly reduce the exposure to data-oriented attacks.

In this paper, we define the notion of *Run-time Scope Enforcement* (RSE) that provides *fine-grained compartmentalization* of data memory within programs. We then describe HardScope, a *hardware-assisted* RSE scheme. HardScope differs from existing defenses in the following important ways: a) it provides *complete meditation* of all variables accesses, b) it is *efficient*, incurring only a small performance overhead for embedded benchmarks, and c) it enables *context-specific* policies. This means that the same piece of code can be granted access to different memory locations depending on the context in which the code is executed. Our main contributions are:

- *Run-time Scope Enforcement*: A novel approach for fine-grained **context-specific memory isolation** within programs (Section 3) to defeat data-oriented attacks.
- *HardScope*: An open-source proof-of-concept implementation of hardware-assisted RSE on the RISC-V architecture that demonstrates **efficient memory compartmentalization** (Section 4).
- *Compiler support and APIs*: Compiler support for **protecting static and automatic variables at run-time** (Section 4.3) without requiring any developer input, and a **programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation.

- *Evaluation*: Analysis of RSE security guarantees (Section 5.1), and evaluation of HardScope's hardware area overhead and minimal performance impact (Section 5.2).

2 Adversary Model & Challenges

Adversary Model. We consider a powerful adversary who has full control over the data memory of the target program. This models buffer overflows and other memory corruption vulnerabilities (e.g., an externally controlled format string[1]) that could corrupt any data memory. However, the adversary cannot modify program code (W⊕X protection). Our adversary model is standard for run-time attacks and consistent with Hu et al.'s DOP attacks [15].

Challenges. Our goal is to prevent the above adversary from mounting DOP attacks. Since DOP attacks (similar to many other data-oriented attacks) require the adversary to modify and access data in unintended ways at run-time, these attacks can be prevented by a *run-time enforcement mechanism* that prevents any data access that would not be permitted during a compile-time check by a correct compiler. Designing a solution to meet this goal requires addressing the following significant challenges:

- C1 *Run-time enforcement:* enforcing variable scopes at run-time requires information which is usually only available at compile-time.
- C2 *Multi-granularity enforcement:* the enforcement mechanism must be configurable for any granularity of protection domain (subject) and protected region (object).
- C3 *Context-specific enforcement:* enforcing different permissions on each invocation of the same subject (e.g., each function), to minimize the attack surface following the principle of *least privilege*.
- C4 *Complete mediation:* protection domains cannot be allowed to increase their permissions accidentally or maliciously, and all memory accesses must be checked with only minimal performance impact and memory overhead.

3 Design Overview

The high-level idea of HardScope is to extend the compiler to emit compile-time information about the visibility of variables, and to extend the underlying hardware to use this compiler-supplied information to dynamically create and update a set of memory access rules against which all memory accesses are checked.

Run-time enforcement. Machine code produced from languages such as C and C++ does not include information available to the compiler about variables and code blocks (C1). RSE needs this information to assign in-memory variables to specific *execution contexts*. To bridge this gap between compile-time lexical scope and run-time execution context, we modified the compiler to instrument the program code with special instructions that record which variables may be used by each code block. HardScope introduces an instruction set extension for this purpose (Section 4).

The compile-time components and behavior of HardScope are illustrated in Figure 1. An unmodified source code program (❶) is fed to the compiler (❷), which checks (as usual) that all variable accesses are correctly scoped. Our new *RSE Plug-in* (❸) in the compiler adds HardScope instructions (❹) at particular locations in the binary (e.g., at the start of functions). This results in a fully-functional program binary, instrumented with HardScope instructions that the HardScope hardware uses to create a set of rules against which all memory accesses can be checked at run-time.

Multi-granularity enforcement. We chose function-level compartmentalization as the granularity of isolation, since this is sufficient to mitigate all currently known DOP attacks (Section 5.1). However, RSE can also be implemented at other granularities (Section 4), without changes to the new HardScope hardware (C2).

Context-specific enforcement. Consider the program (❶) in Figure 1: function `C` receives two pointers and copies data from the first pointer to the second. It can be called from either function `A` or function `B` (call graph shown in Figure 2). In benign execution, variables x and y are only used in a *privileged* execution path, where access control checks prevent misuse (e.g., x could be a secret key). Function `B` contains an exploitable vulnerability allowing the attacker to control the pointers passed to function `C`. Since function `C` can be used to copy arbitrary data between two attacker-controlled pointers, this constitutes a DOP gadget. The attacker could use this to bypass the access control checks on variables `x` and `y` by accessing them through the unprivileged execution path.

HardScope prevents this by providing context-specific enforcement, in which different memory access rules can be associated with *each active instance* of a function (C3). To achieve this, the HardScope hardware creates memory access rules dynamically for each individual function invocation, and stores these in a data structure called the *Storage Region Stack* (SRS). The SRS is kept in hardware-isolated *protected memory*; only HardScope instructions can add or remove SRS entries. Each SRS entry defines an area of memory (e.g., the location of a variable) that may be accessed. The SRS is organized into *frames*; each frame contains all the entries for a particular execution context. The topmost SRS frame corresponds to the active execution context. On each memory access, e.g., load or store, HardScope validates that the memory address matches an entry in the topmost SRS frame.

Specifically, HardScope prevents the attack in Figure 2 as follows: The SRS for function `A` (❺) includes variables x and y, and the SRS for function `B` (❻) includes variables i and j (Figure 2). To allow function `C` to access certain variables, the calling function must use a special instruction (Figure 1 ❼) to *delegate* access to a variable to function `C`: e.g., function `A` must delegate access to x and y. For valid delegation, the calling function must already have access to the delegated variables. Even though the attacker can still manipulate the pointers in function `B`, this function does not have access to x and y (no corresponding SRS entries) and hence it cannot delegate access to these variables to function `C`.

4 Implementation

We developed a proof-of-concept hardware implementation of HardScope and integrated it into the open-source RISC-V Pulpino core.[2] HardScope extends the RISC-V instruction set with seven new SRS management instructions, as shown in Table 1. We augmented the GCC compiler to incorporate a proof-of-concept RSE

[1] CWE-134: Use of Externally-Controlled Format String
https://cwe.mitre.org/data/definitions/134.html

[2] http://www.pulp-platform.org/

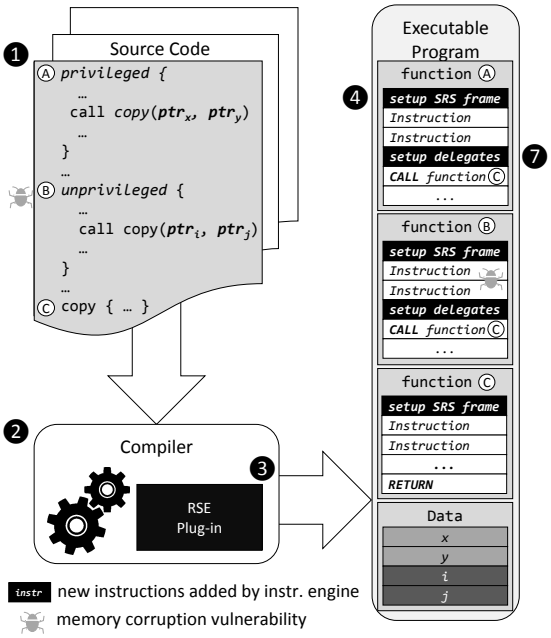

Figure 1: Compile-phase design of HardScope. Run-time memory accesses via pointers ptr_x, ptr_y are limited to variables x and y, while ptr_i, ptr_j are limited to i and j.

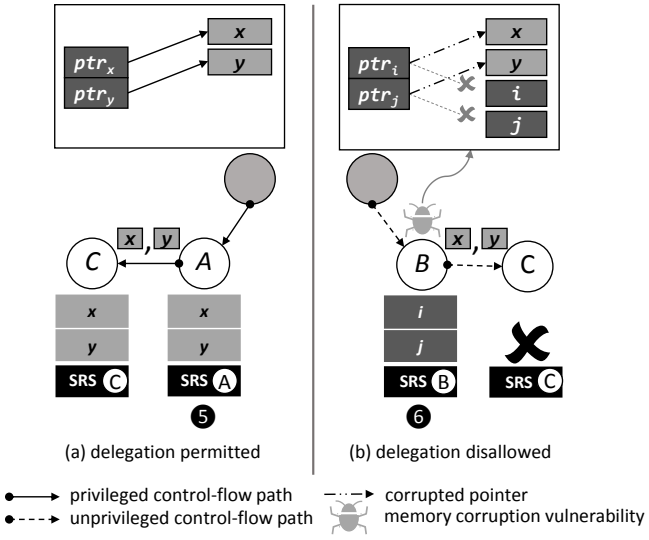

Figure 2: Run-time design of HardScope showing the call graph of program in Figure 1. In (a), access to variables x and y is successfully delegated from A to C. In (b), function B should not have access to x and y, but a memory corruption vulnerability in B is used to corrupt ptr_i and ptr_j to point to x and y instead of i and j. HardScope prevents B from accessing or delegating x and y.

plug-in and a modified RISC-V backend to automatically instrument C programs with the relevant HardScope instructions. These protect static and automatic variables at run-time without requiring any changes to program code. We also developed a HardScope **Programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation. HardScope itself is architecture-agnostic; our choice of RISC-V and Pulpino is due to the open-source nature of the ISA and the RTL implementation, thus enabling us to prototype our solution.

4.1 Instructions

The `sbent` and `sbxit` instructions are used to mark the beginning and end of each execution context. HardScope uses these instructions to track when HardScope is first enabled and when the execution context changes, and thus when new enforcement rules should be loaded in the SRS. `sbent` pushes a new frame on top of the SRS, whilst `sbxit` pops the topmost SRS frame. Program execution starts with an empty SRS and HardScope enforcement is initially disabled. HardScope is enabled by the first `sbent`, and remains enabled until a matching `sbxit` empties the stack.

The `sradd` and `srdda` instructions create an SRS entry in the current (topmost) SRS frame. HardScope uses these instructions to determine the bounds of memory areas that the current execution context is allowed to access. The two operands set the *base* and *limit* address of the storage region respectively. An optional offset is added to to either the limit (`sradd`) or base (`srdda`) register operand.

The `srdel` instruction removes the specified number of SRS entries from the current SRS frame (last in first out). It allows the program to drop unneeded memory access privileges without changing execution context.

The `srdlg` and `srdsub` instructions delegate an SRS entry from the currently executing function either to an invoked callee function or to the caller when the current function returns. HardScope uses these instructions to derive SRS entries for data flows which are not known at compile-time, such as context-specific accesses (Section 3). The operands specify an address to determine which memory address to delegate. The resulting memory address is compared with the current SRS entries and if a match is found, the most recent matching entry is copied to the *next execution context entered*. If the delegation is followed by a `sbent`, the delegated entry is added to the newly created SRS frame. If the delegation is followed by a `sbxit`, the delegated entry is added to the caller's SRS frame.

The `srdsub` instruction is used to delegate a new SRS entry that is a subset of an existing SRS entry. It takes the same operands as `sradd`. If the new subdivided memory region is a subset of an existing SRS entry in the current SRS frame, a new SRS entry is created for a *sub-region* using the new base and limit.

If no matching entry is found in the SRS when `srdlg` or `srdsub` execute, no entry is delegated. This prevents the use of `srdsub` to elevate the access rights of the next execution context beyond the rights of the current, but allows the delegation instructions to be applied to pointers which are not dereferenced directly in the current context. These include null-pointers and intentionally created out-of-scope pointers (e.g., via the use of pointer arithmetic) that are passed to callees for which they are in scope (e.g., accessor functions that receive opaque pointers as arguments).

Table 1: HardScope Instructions. *Mnemonic* is used to refer to the instruction elsewhere in the paper. *Name* is the full name of the instructions. *Operands* lists valid combinations of operands: `rn` is a register, `imm` is an immediate value, and `imm(rn)` is a register to which an immediate offset is added. *Cycles* indicates the number of cycles consumed at execute stage.

Mnemonic	Name	Operands	Cycles
`sbent`	**s**cope **b**lock **ent**er	n/a	1 (+ N)
`sbxit`	**s**cope **b**lock e**xit**	n/a	1 (+ N)
`sradd`	**s**torage **r**egion **add**	`r1, imm(r2)`	1
`srdda`	**s**torage **r**egion **dda** (reverse add)	`imm(r1), r2`	1
`srdel`	**s**torage **r**egion **del**ete	`imm(r1)` `imm`	1 (+ 1)
`srdlg`	**s**torage **r**egion **dele**gate	`imm(r1)` `imm`	1 (+ 1)
`srdsub`	**s**torage **r**egion **d**elegate **sub**-region	`r1, imm(r2)`	1 (+ 1)

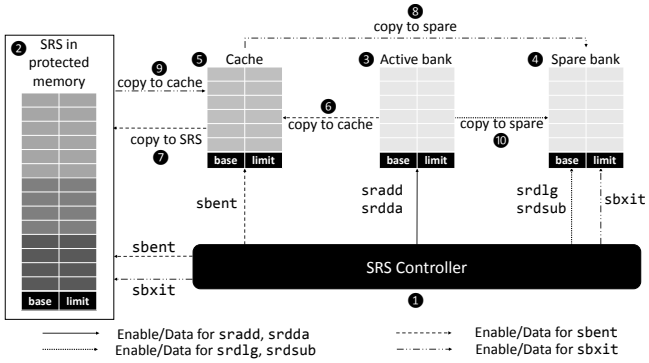

Figure 3: HardScope hardware architecture.

4.2 Hardware Implementation

We modified the instruction decoding stage of the processor pipeline to interpret the new instructions (Section 4.1). After decoding, the appropriate control signals are sent to the HardScope unit, which realizes the execute stage of the new instructions. Figure 3 shows the main components of the HardScope unit: the *SRS controller* (❶), dedicated memory to hold the SRS (❷), and three register banks (❸, ❹, ❺). The *active bank* (❸) holds the entries in the SRS frame for the current execution context enabling each memory access to be compared against *all active entries* efficiently. The *spare bank* (❹) holds entries delegated via `srdlg` and `srdsub` before a HardScope context switch occurs. It allows delegated entries for the *next execution context* to be accumulated ahead of time. When a HardScope context switch occurs, the spare bank becomes the active bank (and vice versa), thus activating the delegated entries. The third bank (❺) is used as a cache to hold a copy of the topmost frame of the SRS. This reduces the latency when the topmost SRS frame is transferred between the stack memory and the spare bank.

When executing `sbent`, the controller activates the spare bank and transfers the contents of the currently active bank to the cache (❻) in a single cycle. The bank that held the previously active frame becomes the spare, and can be used for subsequent delegations. The entries in the cache must be stored for future use, and are transferred to the SRS in protected memory (❼) over at most N subsequent cycles, where N is the maximum number of entries in the cache. During this time, the CPU continues to execute subsequent instructions normally until a new HardScope context switch occurs. Only if a HardScope context switch occurs before the cache has been emptied does the processor stall until the transfer is complete.

When executing `sbxit`, the controller copies the SRS frame from the cache into the spare bank (❽) while retaining delegated entries (i.e., activating the entries that are already in the spare bank). The SRS frame in the previously active bank is no longer needed and is discarded. This executes in a single cycle. The cache, which now holds an out-of-date copy of the active frame, is updated with the topmost SRS frame from the protected memory (❾), which takes at most N cycles, where N is the number of entries in the topmost SRS frame in memory. This does not stall the processor unless another `sbxit` is encountered before the cache is fully populated, in which case the CPU stalls until the next frame is available. However, if an `sbent` is encountered before the cache is fully populated, the partial cache is discarded and replaced with the contents of the active bank, without stalling.

The `sradd` and `srdda` instructions always operate on the active bank. When executing `srdsub`, the controller checks the active bank for an entry containing the given memory region and, if found, adds the new sub-entry to the spare bank. Similarly, in `srdlg`, the controller checks for the matching entry in the active bank and, if found, copies the entry to the spare bank (❿). The `srdlg` and `srdsub` instructions require an additional cycle only if followed immediately by a `sbent` or `sbxit`.

Integrating HardScope into the processor pipeline also required modifying the memory access stage to intercept all memory access requests to the load/store unit. At each `load` or `store` instruction, the requested memory address and the number of requested bytes (one byte, half-word (two bytes), or word (four bytes)) are intercepted and forwarded to the SRS controller, which compares it against all entries in the active bank. The registers in each bank are wired to comparators such that all entries in the bank are checked *in parallel*. If a match is found, i.e. the requested address range is a subset of any of the active entries, then the memory access is granted by the processor's load/store unit, otherwise a hardware fault exception is raised. We design and integrate HardScope to the processor pipeline such that no additional clock cycle latency is incurred to the baseline `load` and `store` instructions.

4.3 Software Instrumentation

Our RSE GCC plug-in and the modified RISC-V backend currently supports automatic instrumentation of C programs at *function granularity* to protect the 1) call stack frame including local variables, return address and other return state information, 2) arguments passed on the stack, 3) heap objects, and 4) global and static variables. The beginning of each distinct execution context is marked by inserting a single `sbent` instruction at the function call site just before the jump instruction. The end of an execution context is marked by inserting an `sbxit` instruction just before the return in the callee function. In Section 5 we show that function-level isolation is sufficient to mitigate all currently known DOP attacks.

However, RSE can also be implemented at other granularities, without changes to the HardScope instructions, by inserting `sbent` and `sbxit` instructions around the instructions that comprise a distinct execution context.

4.4 HardScope Programmer's API

Deeply Nested Pointers. The HardScope Programmer's API enables the handling of code that uses deeply nested pointers e.g., traversing linked lists. This type of code is a challenge for automated instrumentation because e.g., passing the head of a linked list to a function that iterates through the list would require delegation of an SRS entry for each element of the list. Since the number of SRS entries (per frame) is constrained by the HardScope hardware (see Section 4.2), this leads to suboptimal use of HardScope hardware resources and an increased cost in HardScope context switches due to more frequent stalls at run-time. Instead, we propose a programming pattern using the HardScope Programmer's API where one `sradd` instruction is added before the dereference of member pointers to linked member elements, and one `srdel` is added after the dereference. This enables effective yet secure traversal of linked lists and other data structures containing nested pointers.

Heap object allocation. We implemented a wrapper on top of the C standard library `malloc()` function that creates SRS entries for heap allocations, and delegates these to the caller. Other library functions can be similarly wrapped to allow HardScope-instrumented code to be linked against uninstrumented libraries.

5 Evaluation

HardScope meets the stated challenges (Section 2) as follows:

C1 **Run-time enforcement.** The RSE GCC Plug-in infers and emits the necessary HardScope instructions to manage the SRS for stack and global data, as well as dynamic allocations that follow a well-defined pattern. The HardScope Programmer's API allows handling code that is not automatically instrumentable, e.g., uses deeply nested pointers.

C2 **Multi-granularity enforcement.** HardScope can enforce policies with either coarser or finer granularity of execution contexts with the appropriate instrumentation (C2). For instance, HardScope can isolate the function prologue and epilogue from the function body, and protect return addresses on the stack from memory errors in the function body to prevent control-flow hijacking.

C3 **Context-specific enforcement.** In HardScope, the active SRS entries can differ between different invocations of the same subject, depending on which entries have been delegated to this subject (e.g., variables passed to a function by its caller or callee).

C4 **Complete mediation.** HardScope hardware checks every memory access against the active set of SRS entries; accesses without matching entries will fail. Therefore only compiler-admissible memory accesses are allowed.

Instructions that create rules at run-time could potentially be used as *confused deputies*. In a *confused deputy attack*, the attacker attempts to subvert the RSE property by misusing existing HardScope instructions at run-time to create unintended rules. Our design ensures that no such instructions are available to the attacker. Rules for static allocations (stack and global variables) are encoded directly into the instructions. Since these cannot be modified at run-time, they cannot be used as confused deputies.

Instructions that create rules that are determined at run-time are found within memory allocators, e.g., `malloc()`, or code that deals with deeply nested pointers, e.g., iterators annotated using the HardScope Programmer's API. It is reasonable to assume that memory allocators are trusted (or at least that allocations are not influencable by the attacker). We recommend that manually annotated code is vetted for allocators that create rules at run-time. Furthermore, an attacker can only initiate a confused deputy attack if he already controls some part of the code, which is very difficult since every memory access in the instrumented program is checked by the HardScope hardware.

5.1 Security Evaluation

We replicated the DOP attack by Hu et al. [15] and ported the code to Pulpino to evaluate the effectiveness of HardScope. Although it was not possible to port the complete victim ProFTPD server to our FPGA testbed, we focussed on the vulnerable `sreplace()` function [15]. All enforcement rules in our experiments are derived *without any developer annotations* – the GCC intermediate representation contains all information necessary for compile-time instrumentation, including: stack-frame sizes, global variable accesses, function calls, parameters, and return values. Function-granularity isolation is sufficient to prevent the attack.

We verified experimentally *four* ways in which RSE prevents this DOP attack: 1) it prevents the initial memory violation in `sreplace()` as it enforces the indended bounds of input and output buffers when operated on by an unsafe string copy operation (`strncpy()` with incorrect buffer length), 2) it prevents the attack from keeping internal state in unused areas of the program's data section, 3) it denies access to global variables which are accessed by the attack out of their normal context, 4) it denies access to static variables which should only be accessible by code wihin the same compilation unit. We discuss each of these in detail in the extended version of this article [22]. Any one of these would be sufficient to stop the attack, and thus the existence of four distinct mitigations demonstrates the effectiveness of RSE's layered defense strategy.

5.2 Performance and Area Evaluation

Performance overhead. We ran CoreMark[3], a standard performance benchmark for embedded systems, with varying iteration counts on a HardScope-augmented Pulpino synthesized on a Xilinx Zynq-7020 ZedBoard. We observed an average overall performance overhead of 3.2% compared to the execution of unmodified CoreMark on the unmodified Pulpino SoC. All instrumentation in CoreMark was automatically generated by our extended GCC compiler resulting in the binary size increasing by 11%. The number of entries required per SRS frame varied throughout execution between 1 and 23. The overall maximum SRS size was 71 entries in 11 frames, resulting in a memory overhead of 573 bytes (64 bits per entry + 4 bits per frame to record the number of entries).

Area and memory utilization. The area utilization depends primarily on the size of active, spare and cache banks (i.e., the number

[3] http://www.eembc.org/coremark/faq.php

of entries per frame). All three banks are mapped to logic to guarantee single-cycle access parallel checking of all frame entries. The area utilization increases linearly as the number of entries configured per frame increases (for a fixed number of frames), since more entries must be checked in parallel. For a protected memory size of 8 entries × 16 frames, HardScope utilizes 4, 572 LUTs, 1, 760 registers, and one 36 kB block RAM (RAMB36). For a 32 entries × 16 frames configuration (required for the CoreMark performance evaluation above), HardScope utilizes 30, 520 LUTs, 6, 362 registers, and one 36 kB block RAM (RAMB36).

6 Related Work

Various software-only and hardware-assisted memory safety technologies have been proposed and/or deployed (e.g., [2–6, 8, 11, 13, 17–19, 24, 25]). We discuss approaches that aim to mitigate data-oriented attacks in detail in the extended version of this article [22].

Software-only defenses (e.g. DFI [6] and SoftBound [20]) can offer strong security guarantees, but their usefulness is limited by high performance overhead, and by requiring changes to the system software architecture. Consequently, the granularity of enforcement of deployed defenses are often relaxed in favor of improved performance. Memory-safe dialects of C (e.g., CCured [21], Cyclone [16], and Checked C [12]) retrofit C with compile- and/or run-time checks that prevent memory errors from occuring. However, such dialects only benefit programs which are modified to make use of enhanced language features, also incur considerable run-time overhead [16, 21], or preclude certain C features [12].

Hardware-assisted defenses (e.g., BIMA [19], HDFI [27], and CHERI [28]) promise to drastically improve the performance overhead compared to software-based defenses. However, recent advances in attacks against bounds-checking approaches [14] suggest that low-fat pointer schemes which enforce allocation bounds rather than object bounds, such as BIMA [19] are exploitable. On the other hand approaches that track object bounds in separate storage, e.g., Intel MPX [23], HardBound [11], are not faster nor more memory effient than sofware-based approaches. Hardware-assisted tagged memory allow efficient enforcement of memoru access policies, but unlike HardScope only support a small number of simultaneour protection domains (e.g. two domains in HDFI [27]). CHERI [28] is a hardware-assisted capability model that can support various protection models, but requires program re-engineering.

Run-time attestation schemes [1, 9, 10, 29] can only detect, but not prevent, control-flow and non-control-data attacks.

Although HardScope shares many of the same goals as the above schemes, it differs in several fundamental aspects. Compared to software-based schemes (e.g., DFI [6] and SoftBound [20]), HardScope has significantly lower overhead, does not require whole-program static analysis, and can enforce context-specific policies for individual function invocations. HardScope RSE policies can be instantiated for a large class of programs without additional input from developers (cf., YARRA [24]), or software re-engineering (cf., CHERI). HardScope reduces the metadata needed at execution time to the rules for active execution contexts. Active rules are cached in on-chip memory, to enable access checks with no overhead.

7 Conclusion

By implementing and evaluating HardScope, we demonstrated that RSE is an effective approach to protect against data-oriented attacks. HardScope can also enforce memory isolation at coarser or finer granularity, to enable different memory protection strategies.

We provide 1) our enhanced GCC compiler; 2) instrumented binaries of our test programs; and 3) a RISC-V simulator with support for HardScope instructions at https://goo.gl/TAjLxy.

Acknowledgments

This work was supported by the German Science Foundation CRC 1119 CROSSING projects S2 and P3, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant nr. 643964 (SUPERCLOUD), Business Finland under grant nr. 3881/31/2016 (CloSer), Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] Tigist Abera et al. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proc. ACM CCS '16*. 743–754.
[2] Periklis Akritidis et al. 2008. Preventing Memory Error Exploits with WIT. In *Proc. IEEE S&P '08*. 263–277.
[3] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proc. DIMWA '08*. 1–22.
[4] Cristian Cadar et al. 2008. *Data Randomization*. Technical Report MSR-TR-2008-120. Microsoft Research.
[5] Miguel Castro et al. 2009. Fast Byte-granularity Software Fault Isolation. In *Proc. ACM SOSP '09*. 45–58.
[6] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proc. USENIX OSDI '06*. 147–160.
[7] Shuo Chen et al. 2005. Non-control-data Attacks Are Realistic Threats. In *Proc. USENIX Security '05*. 12–12.
[8] Long Cheng, Ke Tian, and Danfeng (Daphne) Yao. 2017. Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. In *Proc. ACM ACSAC '17*. 315–326.
[9] Ghada Dessouky et al. 2017. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *Proc. ACM/EDAC/IEEE DAC '17*. 24:1–24:6.
[10] Ghada Dessouky et al. 2018. LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution. In *ICCAD '18*.
[11] Joe Devietti et al. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proc. ACM ASPLOS '08*. 103–114.
[12] Archibald Samuel Elliott et al. 2018. Checked C: Making C Safe by Extension. In *Proc. IEEE SecDev '18*. 53–60.
[13] Úlfar Erlingsson et al. 2006. XFI: Software Guards for System Address Spaces. In *Proc. USENIX OSDI '06*. 75–88.
[14] Ronald Gil, Hamed Okhravi, and Howard E. Shrobe. 2018. There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection. In *Proc. IEEE SecDev '18*. 102–109.
[15] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proc. IEEE S&P '16*. 969–986.
[16] Trevor Jim et al. 2002. Cyclone: A Safe Dialect of C. In *Proc. USENIX ATC '02*. 275–288.
[17] Dmitrii Kuvaiskii et al. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proc. ACM EuroSys '17*. 205–221.
[18] Volodymyr Kuznetsov et al. 2014. Code-pointer Integrity. In *Proc. USENIX OSDI '14*. 147–163.
[19] Albert Kwon et al. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proc. ACM CCS '13*. 721–732.
[20] Santosh Nagarakatte et al. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proc. ACM PLDI '09*. 245–258.
[21] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proc. ACM POPL '02*. 128–139.
[22] Thomas Nyman et al. 2017. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement. https://arxiv.org/abs/1705.10295
[23] Oleksii Oleksenko et al. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. https://arxiv.org/abs/1702.00719.

[24] C. Schlesinger et al. 2011. Modular Protections against Non-control Data Attacks. In *Proc. IEEE CSF '11*. 131–145.
[25] Konstantin Serebryany et al. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC '12*. 309–318.
[26] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. ACM CCS '07*. 552–561.
[27] C. Song et al. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *Proc. IEEE S&P '16*. 1–17.
[28] Jonathan Woodruff et al. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. IEEE ISCA '14*. 457–468.
[29] Shaza Zeitouni et al. 2017. ATRIUM: Runtime Attestation Resilient Under Memory Attacks.. In *ICCAD '17*.