HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement

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

Ghada Dessouky
Technische Universität Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Shaza Zeitouni
Technische Universität Darmstadt, Germany
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
The widespread use of memory unsafe programming languages (e.g., C and C++), especially in embedded systems and the Internet of Things (IoT), leaves many systems vulnerable to memory corruption attacks. A variety of defenses have been proposed to mitigate attacks that exploit memory errors to hijack the control flow of the code at run-time, e.g., (fine-grained) ASLR or Control Flow Integrity (CFI). However, recent work on data-oriented programming (DOP) demonstrated the possibility to construct highly-expressive (Turing-complete) attacks, even in the presence of these state-of-the-art defenses. Although multiple real-world DOP attacks have been demonstrated, no suitable defenses are yet available.

We present run-time scope enforcement (RSE), a novel approach designed to mitigate all currently known DOP attacks by enforcing compile-time memory safety constraints (e.g., variable visibility rules) at run-time. We also present HardScope, a proof-of-concept implementation of hardware-assisted RSE for the new RISC-V open instruction set architecture. We demonstrate that HardScope mitigates all currently known DOP attacks at multiple points in each attack. We have implemented HardScope in hardware on the open-source RISC-V Pulpino microcontroller. Our cycle-accurate simulation shows a real-world performance overhead of 7.1% when providing complete mediation of all memory accesses.

CCS CONCEPTS
• Security and privacy → Malware and its mitigation; Embedded systems security; Hardware-based security protocols;

1 INTRODUCTION
Memory corruption and memory disclosure vulnerabilities are still a persistent source of threats against software systems, although known for over two decades. The main problem is that modern software still contains a vast amount of unsafe, legacy code written in memory unsafe programming languages (e.g., C and C++), especially in embedded systems and the Internet of Things (IoT) [12]. The lack of memory safety in these languages and the inevitability of software bugs therefore leaves many systems vulnerable to attacks that exploit memory corruption.

Control-flow attacks, which hijack the execution flow of a program, are well-known, and various defenses against them have been proposed (e.g., [1, 17, 19]). In contrast, non-control-data attacks do not need to modify the control-flow of the targeted program, and thus cannot be prevented by the same defenses. Instead, non-control-data attacks corrupt data used for decision-making, e.g., to leak sensitive data or escalate privileges by corrupting variables used in authorization decisions. Some defenses against non-control-data attacks have also been proposed (e.g., [7, 24]). These defenses provide protection against non-control-data attacks that only target individual pieces of (security-critical) data. However, recent work has shown that non-control-data attacks can be generalized to achieve Turing-complete execution, in what has become known as Data-Orientated Programming (DOP) [14]. In DOP, the attacker carefully corrupts non-control-data to build up sequences of operations (called data-oriented gadgets) without modifying the program’s control-flow. Each gadget simulates a virtual operation on some attacker-controlled input. Unlike previous non-control-data attacks, DOP can be highly expressive (e.g., including assignment, arithmetic, and conditional decisions). Since DOP can reuse virtually any data, it is highly challenging to prevent compared to traditional non-control-data attacks. Hu et al. [14] found a
significant number of DOP gadgets in existing real-world software. Although practical DOP attacks have already been shown against real-world software [11, 14], no suitable defenses have yet been proposed. As practical defenses against control-flow attacks become more widespread [15], it is likely that DOP will become the next appealing attack technique for modern run-time exploitation.

**Goals and Contributions.** Our primary goal is to defend against non-control-data attacks and all currently known DOP attacks. The intuition behind our approach is as follows: Structured programming languages allow developers to define variables with a limited scope of visibility. In block-structured programming languages, such as C and C++, the compiler limits the portion of source code in which a variable can be used. This code block is called the lexical scope of the variable. A correct C compiler will enforce that a function is only able to access variables in its lexical scope. Any scope violation results in a compiler error. The compiler therefore ensures compile-time intra-program isolation. However, all currently known DOP attacks violate intra-program isolation at run-time, since there is no equivalent enforcement of scope. A mechanism that provides variable scope enforcement at run-time would significantly reduce the number of available DOP gadgets.

In this paper, we define the notion of Run-time Scope Enforcement (RSE) that provides intra-program memory isolation. We demonstrate that RSE can mitigate all currently known DOP attacks. We stress that it is not possible to guarantee the absence of DOP gadgets in arbitrary programs.\(^1\) Hence, we do not claim to prevent DOP in the general case.

Hu et al. [14] explain that since DOP gadget are plentiful, any potential mitigation mechanism must provide complete mediation of all variable accesses. Therefore, as we explain in detail in Section 8, existing defenses against non-control-data attacks are too coarse-grained to defend against DOP. Furthermore, existing software-only schemes suffer from high performance and memory overheads [6, 20, 21].

To achieve complete and efficient intra-program isolation, we propose HardScope, a hardware-assisted RSE scheme. HardScope introduces a set of seven new instructions. Compiler-assisted instrumentation places HardScope instructions in the program to ensure that all memory access constraints are also enforced at run-time. As the program executes, these instructions dynamically create rules in the HardScope hardware that define which code blocks can access which pieces of memory. One significant challenge is to minimize the performance overhead of checking these rules on every memory load/store operation. To overcome this challenge, we have designed an efficient method for storing the rules as a stack, such that all rules applicable to the currently executing code block are always at the top of the stack and can be checked simultaneously (Section 5).

Since enforcement rules are created and updated dynamically, HardScope (or any other RSE scheme) enables context-specific memory isolation. 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. For example, if a particular function can be called as part of either a privileged or unprivileged execution path, HardScope can allow/deny it access to certain variables in memory depending on which path was executed\(^2\). Prior defenses against non-control-data attacks do not provide this important feature, which is critical to reducing the number of available DOP gadgets, but instead use static policies [3–6, 10, 27].

As we explain in Section 5.2, HardScope can also mitigate return-oriented programming (ROP) attacks by protecting functions’ return state information on the stack. In Section 7 we describe how RSE can also support other types of intra-program memory isolation at either coarser or finer granularity in order to enforce different types of memory access policies (e.g., protecting legacy binaries).

We have developed a proof-of-concept implementation of HardScope targeting the RISC-V instruction set architecture and toolchain and we enhanced the RISC-V compiler to automatically insert the necessary instructions to protect variables at run-time to mitigate known DOP attacks. We have extended the official RISC-V simulator, Spike, to support our new instructions. Finally, to obtain a cycle-accurate measurements of the real-world performance of our approach, we have implemented HardScope on the open-source Pulpino core\(^3\) running on a Zedboard FPGA\(^4\) (Section 5).

Our main contributions are as follows:

- **Run-time Scope Enforcement:** A novel approach for fine-grained context-specific intra-program memory isolation (Section 4).
- **HardScope:** A proof-of-concept implementation of hardware-assisted runtime scope enforcement on the RISC-V open instruction set architecture to achieve efficient inter-procedural memory isolation capable of mitigating all currently known DOP attacks (Section 5).
- **Automatic Instrumentation:** Compiler support for protecting variables at run-time without requiring any developer input or data-flow analysis (Section 5.2).
- **Evaluation:** An evaluation of the security considerations of RSE (Section 6.1), systematic analyses of how HardScope mitigates published DOP attacks (Section 6.2), and an evaluation of HardScope’s performance impact and hardware area overhead (Section 6.3).

---

\(^{1}\)A pathological case would be a program that contains all necessary gadgets, a gadget dispatcher, and all the data in the same function.

\(^{2}\)RSE can in fact enforce different policies on each distinct invocation of a given function even in a single execution path.

\(^{3}\)http://www.pulp-platform.org/

\(^{4}\)http://zedboard.org/
2 BACKGROUND

Memory errors and bounds checking. Memory errors that can be exploited by an attacker may be used as entry-points to a vulnerable program. These may provide either read or write access to program memory. C and C++ provide no built-in run-time protection against accessing or overwriting data that resides in the program’s own memory space. Modern compilers may insert checks around operations on local, global, and heap objects (e.g., arrays), that verify if run-time whether data written to a memory object is within the boundaries of that object. Such bounds checking can prevent buffer overflows, but requires instrumentation of code and incurs high performance and memory overhead [25], even with hardware support for such bounds checks [23].

W⊕X. By utilizing memory vulnerabilities, an attacker can subvert the integrity of the program memory. Direct modification of program code in modern processors is prevented by W⊕X memory access policies such as DEP [13].

Probabilistic defenses. In sophisticated run-time attacks, the attacker crafts payloads that cause the program to behave in an unintended manner. These payloads usually refer to data and code by their addresses in memory. Attackers can find these addresses by offline analysis of the program memory layout. Address Space Layout Randomization (ASLR) [19] randomizes the memory layout of the program on each execution. ASLR typically randomizes the base address of the executable and the positions of the stack, heap, and libraries. This prevents an attacker from reliably addressing known targets, for instance identifying a particular function to jump to, or reading/modifying a particular variable. However, ASLR defenses are susceptible to information leakage (e.g., by obtaining the value of a well-known pointer), and are routinely bypassed in real-world exploits [26].

Data-Orientated Programming. In DOP [14], an attacker carefully tampers with non-control-data to enable the execution of sequences of operations within the program on attacker-controlled input. This sequence of operations constitutes a data-oriented gadget which, in turn, represents a single virtual machine instruction executing on top of benign program logic. A gadget dispatcher (e.g., an attacker-controlled loop within the benign program), enables the attacker to chain together the execution of an arbitrary number of data-oriented gadgets and achieve expressive computation. DOP has been shown to be a practical exploitation technique in real-world software [11, 14].

3 ADVERSARY MODEL & REQUIREMENTS

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 string6) that could lead to arbitrary corruption of data memory. However, the adversary cannot modify program code at run-time (W⊕X). Our adversary model is similar to previous work on defenses against non-control-data attacks.

Requirements. Given the adversary capabilities defined above, we require a mechanism that prevents this type of adversary from mounting DOP attacks. Therefore, we require a run-time enforcement mechanism that prevents any modifications of control-data and non-control-data that would not be permitted during a compile-time check by a correct compiler. We derive the following requirements for a mechanism that will mitigate all currently known DOP attacks:

R1. Block-oriented enforcement. Prevent individual code blocks from accessing program data if this access would not be allowed by a compile-time check. Conversely, all accesses allowed by the compiler must also be allowed at run-time.

R2. Context-specific enforcement. Enforce different permissions on each invocation of the same code block, to minimize the attack surface following the principle of least privilege.

R3. Complete mediation. Check all memory accesses with only minimal performance impact and memory overhead.

Although not the main focus of this paper, a mechanism that meets these requirements can also be used to defend against ROP, by protecting the function return state (e.g., the return address on the stack) as explained in Section 5.

4 DESIGN OVERVIEW

The purpose of HardScope is to achieve Run-time Scope Enforcement (RSE) – specifically to enforce intra-program memory isolation in order to mitigate current DOP attacks. Fundamentally, this is achieved by instrumenting programs at compile-time, and then using HardScope hardware in conjunction with this instrumentation to enforce memory isolation at run-time.

A challenge in realizing RSE is that binary program code produced from languages such as C and C++ does not include run-time information about variables and code blocks available to the compiler. RSE requires this information to assign memory regions containing variables to specific execution contexts. To bridge this gap between compile-time lexical scope and run-time execution context, we modify the compiler to instrument the program code with special instructions that record the variables operated on by each block of code. HardScope introduces an instruction set extension for this purpose.

The compile-time components and behavior of HardScope are illustrated in Figure 1. An unmodified source code program (1) is fed to the compiler (2), which checks that all variable accesses are correctly scoped. Our new RSE Instrumentation Engine (3) in the compiler adds HardScope instructions (4) 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.

6CWE-134: Use of Externally-Controlled Format String https://cwe.mitre.org/data/definitions/134.html
Run-time behavior and components of HardScope are shown in Figure 2. At run-time, the new instructions tell the HardScope hardware which variables are in the scope of the current execution context: i.e. specific invocations of particular blocks of code that may legitimately operate on those variables. This information is stored in a stack data structure called the Storage Region Stack (SRS). The SRS is stored in hardware-isolated protected memory that can only be modified by HardScope instructions. These instructions add or remove entries from the SRS in order to indicate which variables should be accessible from the current execution context. On each memory access, e.g., load or store, HardScope validates that the memory address matches an entry for the current execution context in the SRS. In effect, HardScope associates each execution context with a whitelist of data it may legitimately access at run-time.

To correctly configure the HardScope hardware, the new instructions provide the following functionality:

1. Specify what storage regions are accessible by an execution context.
2. Allow an execution context to dynamically delegate access to a storage region to another execution context (e.g., during function invocation \( \Phi \) or return).
3. Subdivide a storage region so that partial access can be delegated.

Section 5 describes the HardScope instructions in detail.

Another design challenge arises from the requirement to provide context-specific enforcement (Requirement R2). For example, the sample program in Figure 1 (\( \Phi \)) includes function C that receives two pointers as input and copies data from the first pointer to the second. It is invoked from either function A or function B (the call graph is shown in Figure 2). In benign program execution, variables \( x \) and \( y \) are only used in a privileged execution path, where safety-checks protect them from misuse. 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 data between two attacker-controlled pointers, this constitutes a gadget usable for DOP. In Figure 2, the SRS for function A (\( \Phi \)) includes variables \( x \) and \( y \), and the SRS for function B (\( \Phi \)) includes variables \( i \) and \( j \). By default, HardScope prevents function C from accessing any of these variables. To allow function C to access certain variables, the calling function must use a special instruction (Figure 1 (\( \Phi \)) to delegate access 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. HardScope RSE therefore prevents the attack scenario described in Figure 2: even though the attacker can control the pointers in function B, this function does not have access to \( x \) and \( y \), and hence cannot delegate access to these variables to function C.

Requirement R3 calls for a mechanism that can be used to check every memory access which leads to another challenge: minimize the performance impact and memory overhead of HardScope whilst providing complete mediation of memory accesses. We discuss our solution to this in Section 5.
5 IMPLEMENTATION

In this section, we describe how HardScope works by explaining how it provides inter-procedural memory isolation (i.e., at function-level granularity) since it is sufficient to mitigate all currently known DOP attacks (Section 6). However, HardScope is available at other possible granularities as described in Section 7. Recall that, each active instantiation of a function maintains a separate stack frame on the program’s normal call stack. The stack frame typically contains the return address, other return state information (saved registers etc.), parameters passed on the stack, and data storage for local variables. In addition, a function might operate on global or static variables allocated from the program’s data section, and/or dynamically allocated variables stored in the program’s heap. HardScope is designed to be aware of all of them in order to correctly regulate memory isolation.

As described in Section 4, the HardScope Storage Region Stack (SRS) is conceptually a whitelist of accessible memory areas. Each entry in the SRS consists of the base and limit boundary addresses of a memory area containing program data. We denote all such areas as storage regions. The SRS is organized into frames, where each single frame is occupied by all storage region entries for the same function instance. The structure of the SRS mirrors that of the program call stack, i.e., each frame in the call stack corresponds to a frame in the SRS. The topmost frame in the SRS applies to the currently active function. On each memory access, e.g., load or store instruction, HardScope enforces that the memory address required matches a storage region entry in the topmost SRS frame.

5.1 HardScope Instructions

HardScope extends the RISC-V instruction set with seven new instructions for SRS management, as shown in Table 1. The sbent, sbxit, srse, and srlmt instructions are used to specify storage and code regions. The srdlg and srdlgm instructions enable dynamic delegation of storage regions. The srsub instruction enables subdivision of storage regions.

The sbent and sbxit instructions mark the beginning and end of an execution context. When sbent is executed, a new frame is pushed on top of the SRS. Conversely, sbxit pops the topmost SRS frame.

The srse and srlmt instructions are used to create a storage region entry in the current (topmost) SRS frame (i.e. an SRS entry). The srse sets the base address, and srlmt sets the limit address. These instructions take either a (register + offset) operand, or an immediate value describing an absolute address.

The srsub instruction is used to create an SRS entry that is a subset of another existing SRS entry. It takes two register operands and an immediate value. The first register specifies a new base address, and the (second register + immediate offset) specifies a new limit address. If this memory region is found to be contained within an existing SRS entry in the current SRS frame, a new SRS sub-entry is created using the new base and limit. Otherwise, a hardware fault is raised.

The srdlg and srdlgm 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. They take a single register operand and an immediate offset to specify a memory address to delegate. This memory address is compared with the SRS entries in the current SRS frame and if a match is found, the 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. If multiple matching entries exist in the SRS frame, only the most recently added entry is delegated. If no matching entry is found, a hardware fault is raised.

The srdlg instruction operates in the same way as srdlg, but removes the delegated entry from the current SRS frame. For example, srdlg is used to delegate sub-entries created by srsub, since the current function already has the full entries. Program execution starts with an empty SRS and HardScope enforcement is initially disabled. The first function that supports HardScope, typically the program’s main function, must execute sbent first to enable HardScope. HardScope remains enabled until a matching sbxit that empties the stack is executed.

5.2 RSE Instrumentation

We modified the RISC-V GNU Compiler Toolchain to use HardScope to automatically protect 1) the return address and other return state, 2) stack variables, 3) arguments passed on the stack, 4) heap objects, and 5) global and static variables in C programs at function granularity. By convention, the compiler adds a function prologue to the beginning, and a function epilogue to the end of each function. The purpose of the prologue is to prepare the run-time environment for function execution. Conversely, the epilogue restores the stack and registers to the state they were in before the function call and finally returns control to the caller.

Run-time Scope Enforcement (RSE) instrumentation occurs in three phases; 1) SRS entry initialization is added to function prologues, 2) function calls are instrumented to trigger a HardScope context switch (sbent) and to delegate access to any SRS entries needed by the callee, and 3) delegation rules for return values and HardScope context switches (sbxit) are added to function epilogues. We now describe how RSE instrumentation is used within a single function, then how it is applied to function calls and returns, and finally how it is used to protect return address information and heap objects.

Function prologue instrumentation. The function prologue is responsible for allocating space on the call stack for local variables, and storing the return address and old frame pointer to the stack. HardScope instructions are inserted before the standard prologue begins in order to create new SRS entries for local variables allocated by the prologue, as well as for any static or global variables accessed by the function.
Table 1: HardScope Instructions. *Operands* lists the valid combinations of operands for each instruction; †r1 indicates a register and imm indicates an immediate operand. *Cycles* indicates the number of cycles required during the processor execute stage.

| Mnemonic  | Name                  | Operands | Cycles  | Note                                      |
|-----------|----------------------|----------|---------|-------------------------------------------|
| sbent     | scope block enter    | n/a      | 1 (+N)  | at most N additional cycles if followed sbent or sbxit within N executed instructions, N equal to number of storage region entries in cache (Sec. 5.3) |
| sbxit     | scope block exit     | n/a      | 1 (+N)  |                                           |
| srbse     | storage region base  | imm(r1)* | 1 +1    | additional cycle if immediately followed by HardScope instruction, imm offset added to r2 address |
| srlmt     | storage region limit | imm(r1)* | 1 +1    | additional cycle if immediately followed by sbxit |
| srsb      | storage region sub-entry | r1, imm(r2) | 1 +1    |                                           |
| srdlg     | storage region delegate | imm(r1)* | 1 +1    |                                           |
| srdigm    | storage region delegate (move) | imm(r1)* | 1 +1    |                                           |

Listing 1 shows a function prologue (❶) that reserves space for a stack frame containing two 32-bit variables, the return address, and frame pointer (16 bytes in total) from the stack (line 10). It then stores the value of the return address register (ra), and the register holding the frame pointer (a0) to the stack frame (lines 11, 12). During instrumentation, the prologue is prepended with srbse and srlmt instructions (❷) that add a SRS entry covering the whole stack frame (e.g. 16 bytes in Listing 1). The limit for this entry is one less than the current stack pointer value, and the base address is calculated by subtracting the size of the stack frame from the current stack pointer. The compiler already knows the size of the stack frame since this is used to decrement the stack pointer (line 10). The SRS entry must be added before the standard prologue begins so that the prologue can access the stack to store the return address (line 11).

In C programs, global objects can be accessed from any scope. However, during RSE instrumentation, instructions creating SRS entries for global objects are only added to functions that refer directly to these objects7. For example, in Listing 1, a global myobject is accessed from the function’s scope, so a SRS entry for myobject is created before the standard prologue begins (❶). Separate SRS entries are added for each object accessed by the function. The addresses and sizes of global objects are already known to the compiler. Local and global static variables are handled similarly. Conversely, functions that access global objects indirectly (e.g. via function pointers) may need to receive the necessary SRS entries for these global objects via run-time delegation.

Function call and return instrumentation. RSE also requires the beginning and end of each distinct execution context to be marked in the program code. The beginning of an 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. As noted in Section 4, callers may also pass pointers as arguments during function calls. Similarly, a callee may return a pointer to the caller.

---

7 This could also be done at finer granularity, as explained in Section 7.
Listing 3: Instrumented function returning a pointer via argument register a0.

Listing 3 shows an instrumented version of a function returning a pointer via argument register a0. We divided the function into a prologue, which creates the SRS entry for the return object, and an epilogue, which returns control to the caller. The call to the malloc() function allocates memory on the stack, and the memcpy() function copies the data into the allocated memory. The result is stored in a 32-bit register, which is then returned to the caller.

Listing 4: Instrumented function with return state protection. The function prologue and epilogue share the same execution context.

Listing 4 demonstrates how our full RSE instrumentation protects the return address and saved frame pointer against modification by potentially vulnerable code within the function body. Since the prologue and epilogue surround the function’s main body, we instrument these to execute in two distinct execution contexts. The prologue (❶) and epilogue (❷) share their execution context and SRS frame, and can thus access the same area of the stack. They can also access argument objects passed by reference and delegated by the caller, and delegate these forward to the function body (❸). Once the execution context of the function body has been entered (line 11) an additional SRS entry is created for local stack variables (❹), before the execution of the main function body begins (❶). Exiting the function body’s execution context returns to the execution context set up by the prologue, whilst control flow proceeds to the epilogue (❷). This protects the return state information during the execution of the function body, and thus mitigates attacks that require modification of this control-flow information (e.g. ROP).

Heap object allocation. We implemented a wrapper on top of the C standard library malloc() function that creates SRS entries for the heap memory allocated by malloc(), and delegates these to the caller.

5.3 Hardware Implementation

We implemented HardScope as an extension to the open-source RISC-V Pulpino core. Integrating HardScope into the processor pipeline required modifying the instruction decoding stage to interpret the new instructions and trigger the appropriate control signals to the HardScope unit. Figure 3
shows the hardware components of HardScope: the SRS controller (❶), a dedicated stack memory to hold the SRS (❷) and three register banks (❸, ❹, ❼). The active bank (❸) holds the SRS entries for the currently active SRS frame. The spare bank (❹) holds SRS entries delegated via srdlg and srdlgm before a HardScope context switch occurs. When a HardScope context switch occurs, the spare bank becomes the active bank (and vice-versa), thus activating the delegated SRS entries. The third bank (❹) is used as a cache to hold a copy of the topmost frame of the SRS.

When executing sbent, the controller activates the spare bank and transfers the contents of the currently active bank to the cache (❺), that executes in a single cycle. When the transfer completes, the bank that held the previously active frame becomes the spare, and can be used for subsequent delegations. The SRS entries in the cache are transferred to the SRS in protected memory (❹) over at most \( N \) subsequent cycles, where \( N \) is the maximum number of SRS 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 (❹) while retaining any delegated SRS entries (i.e. activating the entries that are already in the spare bank). The SRS frame in the previously active bank 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 (❷), which takes at most \( N \) cycles, where \( N \) is the number of SRS entries in the topmost SRS frame in memory. This transfer to the cache 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 a 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 processor.

The sbse, srlmt and srsub instructions always operate on the active bank. When executing srsub, the controller checks the active bank for a SRS entry containing the given memory region and, if found, adds the new sub-entry to the active bank. Similarly, in srdlg/srdlgm, the controller checks for the matching SRS entry in the active bank and, if found, copies/moves the entry to the spare bank (❹). The srdlgm and srsub instructions each require a single cycle, but will also update the active bank in the following cycle if a matching entry is found. Thus, if either of these instructions is followed immediately by another HardScope instruction that also modifies the active bank, the processor may stall for at most one cycle.

Integrating HardScope into the processor pipeline also required modifying the memory load/store stage to intercept memory accesses and enforce the desired memory isolation. At each load or store instruction, the requested memory address is forwarded to the SRS controller that compares them with all SRS entries in the currently active bank in simultaneously. The registers in each bank are wired into comparators that enable each entry to be checked simultaneously, thus avoiding the introduction of additional overhead to the processor pipeline. When a match is found, access to the memory address is granted to the processor load/store unit. The overhead of each HardScope instruction is summarized in Table 1.

### 5.4 Simulator Implementation

We extended Spike\(^8\), the official RISC-V ISA simulator to support our HardScope instruction set extension. Spike, being part of the official RISC-V infrastructure, is currently the most accurate simulator for RISC-V assembler programs. It is regularly maintained by the RISC-V community, well

\(^8\)https://github.com/riscv/riscv-isa-sim
integrated into the toolchain, and supports debugging with the GNU debugger (GDB).

We used it to analyze the security properties and performance profile of RSE, and we have also made it available for developers and researchers who wish to reproduce our results or experiment with other uses of our HardScope instruction set extension.9

The simulator was extended by adding a SRS module to the Memory Management Unit (MMU) of the processor. Similar to the hardware design described in Section 5.3, this module includes two banks of SRS registers for active and delegated entries, and a stack for inactive frames. Each executed HardScope instruction is passed to our SRS module, which faithfully simulates the behavior of a real hardware implementation. Our SRS module also collects performance profiling statistics including the number of executed instructions, frequency of context switches, sizes of SRS frames, and the number of access checks performed.

6 EVALUATION

We describe the security considerations of HardScope with reference to the requirements defined in Section 3, explain how it mitigates one of the published DOP attacks from [14], and finally analyse its performance and area overhead.

6.1 Security Considerations

R1. Block-oriented enforcement. As described in Section 4, the compiler places HardScope instructions in the code to create the necessary SRS entries for the program to run. These entries thus mirror the compile-time memory access constraints. The instructions are placed such that they cannot be misused by an attacker to create additional SRS entries. Since the HardScope hardware checks every memory access against the currently active set of SRS entries, a memory access without a matching entry will fail. Therefore, the only possible memory accesses are those that would be allowed by a compile-time check. Conversely, the compiler can always ensure that the relevant SRS entries will be present at run-time, thus allowing all legitimate memory accesses. Dynamically allocated heap objects require special consideration since their addresses are only known at run-time. We require assistance from a trusted memory allocator when creating SRS entries for heap objects, since a malfunctioning or attacker-controlled allocator might perform double allocation or allocate memory regions outside the heap area. However, determining the correctness of memory allocators is an orthogonal problem.

R2. Context-specific enforcement. This is achieved by the delegation instructions in HardScope. The set of active SRS entries can, therefore, differ between different invocations of the same code block depending on which entries have been delegated to the block by its caller or callee functions.

R3. Complete mediation. Every memory access in the instrumented program is checked against the active set of SRS entries by the HardScope hardware. The performance and area overhead are discussed in Section 6.3.

DOP attacks. As shown above, HardScope fulfills all requirements defined in Section 3 to mitigate currently known DOP attacks. In Section 6.2 and Appendix C, we present step-by-step analyses of how HardScope mitigates each of these DOP attacks.

ROP attacks. Additionally, HardScope also defends against ROP attacks. As explained in Section 5, the function prologue and epilogue are placed in a separate execution context from the function’s main body. By assigning the function’s return state information only to the prologue/epilogue’s execution context, HardScope protects this information from the potentially corrupted main body of the function. Without the ability to control the function’s return value, the attacker cannot mount ROP attacks.

6.2 Example Use Case

Hu et al. [14] demonstrated three practical end-to-end DOP attacks against the ProFTPD file server. In this section we explain how HardScope mitigates the most detailed of these attacks, and in Appendix C we explain how it mitigates the other DOP attacks, including the attack by Evans [11].

Each attack in [14] exploits the same stack buffer overflow vulnerability in a general purpose string replacement function (CVE-2006-581510). An excerpt from the vulnerable sreplace() function is shown in Appendix A. The vulnerability allows the attacker to read and write arbitrary locations in memory. This is consistent with our adversary model in Section 3.

To show how HardScope mitigates this attack, we first describe the steps of the attack in detail. The attacker’s goal is to leak the server’s OpenSSL private key. Using ASLR, the program randomizes the memory location at which this key is stored, so direct access is unlikely to succeed. Instead, the attacker constructs a virtual DOP program that first obtains a pointer to the OpenSSL context structure (ssl_ctx) from a well-known location, and then dereferences a chain of seven pointers to correctly determine the location of the private key.

This attack requires access to three different types of data-oriented gadgets: assignment, addition, and pointer dereferencing. The assignment gadget is constructed from the vulnerable sreplace() function. The addition gadget is realized by corrupting two integer fields in a global data structure, the session.total_bytes_out and session.xfer.total_out, and performs the operation session.total_bytes_out += session.xfer.total_out. The dereference gadget is built by corrupting a string pointer in another global data structure, the main_server->ServerName. This dereferences the pointer main_server->ServerName and copies the result to a known position in a global static buffer resp_buf.

9https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2006-5815
We replicated this DOP attack and ported the code to RISC-V to evaluate the effectiveness of HardScope. Although it was not possible to port the complete ProFTPD to our FPGA testbed or simulation environment, we concentrated our evaluation on the vulnerable `sreplace()` function.\textsuperscript{11} We manually instrumented this code with HardScope instructions according to the scheme described in Section 5.2. We used our modified Spike simulator to trace the execution for both benign and malicious inputs. We verified that our instrumentation did not affect the correctness of the program under benign inputs. The source code and instrumented symbolic assembler files are included in the supplementary material.\textsuperscript{12}

We identified five ways in which RSE prevents this DOP attack. Any one of these would be sufficient to stop the attack, and thus the existence of five distinct mitigation points demonstrates the effectiveness of RSE’s layered defense strategy. All five were also verified experimentally using our modified Spike simulator.

1). The initial memory violation in `sreplace()` is caused by an incorrect string length returned by `strlen()` when invoked on an attacker-controlled string `pbuf` without a trailing null terminator. The faulty length is used in a call to `strncpy()`, which moves the contents of the string to a local stack buffer `buf`. The maximum number of bytes copied should be bounded by the size of `buf`, but the bound is calculated as `sizeof(buf) - strlen(pbuf)`, which yields a negative value. When this value is interpreted as an unsigned integer by `strncpy()`, it causes a buffer overflow. As described in Section 5.2, our RSE instrumentation records and enforces the originally-intended bounds of `buf` and `pbuf`, thus preventing the out-of-bounds access by `strlen()` and `strncpy()`.

2). The DOP program keeps its internal state in the program’s memory at locations that allow the gadget dispatcher to reach and chain together data-oriented gadgets. In this attack, the DOP program stores its data in an unused area of the data section. By default, RSE denies access to such unused areas from all functions. The attacker could attempt to work around this by using pre-existing global variables. However, this means that all the DOP gadgets must either share access to the same global data structure, or all be reachable by data flows to and from this data structure. However, gadgets that legitimately share access to such data are more likely to use this data in the benign program. This is undesirable for DOP because corrupting or re-purposing such data could have unwanted side-effects on program execution, or the data could be overwritten by other gadgets, thus significantly limiting the amount of data or the set of gadgets that can be used in this attack.

3). The exploit corrupts variables in global data structures to control the operands of the addition and dereference gadgets. During benign execution, the `sreplace()` function should only operate on a copy of the `main_server.ServerName` pointer passed by value, and on unrelated fields in the global `session` structure, also passed by value. Therefore, our compiler does not give `sreplace()` access to these global variables, and thus RSE blocks the addition and dereference gadgets by preventing access to their respective operands.

4). The dereference gadget accesses the `ssl_ctx` via a pointer included in the DOP payload, which must therefore be known a priori by the attacker. Since `ssl_ctx` is defined as a global static in the `mod_tls.c` source file, RSE prevents the dereference gadget from accessing this structure and linked structures. The DOP program would otherwise traverse the chain of linked structures to determine the location of the secret key.

5). Once the address is known, `sreplace()` is used to corrupt a local static `mons` array containing string pointers. The `mons` array (also shown in Appendix B) contains pointers to string literals in the program’s data section that are used by the `pr_strtime()` function to format a `time_t` value to a human readable representation. Each pointer in `mons` is redirected to the same memory location. One byte of the secret key is then copied to that location. Whenever any `time_t` value is formatted by the corrupt `pr_strtime()` function, it includes a few bytes of the key. These strings are sent to the user, who in this case is the attacker. The attacker repeats this process for subsequent bytes until the entire key has been extracted. RSE prevents this exfiltration in two ways: Firstly, because the scope of `mons` is local to the `pr_strtime()` function, `sreplace()` cannot overwrite it with new pointers. Secondly, RSE ensures that `pr_strtime()` cannot access the key, even if it attempts to dereference corrupt `mons` pointers.

Although RSE cannot guarantee the absence of usable data-oriented gadgets in arbitrary programs, this example demonstrates how RSE significantly limits the expressiveness of DOP attacks in programs with at least some degree of structural data separation (as is the case in virtually all real-world programs).

6.3 Performance and Area Overhead

We extended the open-source RISC-V Pulpino SoC\textsuperscript{13} with our HardScope instruction set extension. To verify the functionality and performance impact of HardScope, we performed cycle-accurate simulations of the extended core using ModelSim. For area utilization, we evaluated the extended core using Xilinx Vivado\textsuperscript{14}. Our performance evaluation was performed on the same test program used for our security evaluation (i.e., the ported ProFTPD code described above). It consisted of a total of 519 lines of C code, with the deepest call chain being seven levels deep.

**Performance evaluation.** We measured the overall performance impact of HardScope by performing a cycle-accurate simulation of our test program. Table 2 shows the total cycle

\textsuperscript{11} For the same reason, it is also not possible to run standard SPEC benchmarks on this small microcontroller.

\textsuperscript{12} http://www.pulp-platform.org/

\textsuperscript{13} https://www.xilinx.com/products/design-tools/vivado.html
We instrumented the Spike simulator to profile the number was 25 entries, resulting in a maximum memory overhead of 204 bytes (64 bits per entry + 4 bits per frame to record the cache operations triggered by \texttt{sbent} and \texttt{sbxit}. As explained in Section 5, the cache operations only cause the processor to stall if another \texttt{sbent} or \texttt{sbxit} is encountered before the cache has been written to/read from the protected memory. We instrumented the Spike simulator to profile the number of SRS entries in the cache at each \texttt{HardScope} context switch. Figure 4 plots the number of entries in the SRS cache against the duration in cycles of the execution context during which the cache transfer occurs. For clarity, this plot only includes durations of 16 cycles or less. Points on or above the line result in stalls. Our profiling shows that, in the test program, 10.6% of \texttt{HardScope} context switches result in processor stalls. However, 83% of these stalls incur only one additional cycle. Points at \( y = 0 \) indicate leaf functions for which the corresponding SRS frame is never flushed into protected memory.

### Area and memory overhead

We synthesized \texttt{HardScope} using Xilinx Vivado targeting a Zedboard (hosting a Virtex-7 XC7Z020 FPGA) and measured an area utilization of 7804 slice LUTs and 3196 slice registers and one 18Kb BRAM for the SRS. In our implementation, we configured the maximum depth of the SRS to be 256 64-bit entries (16 frames each holding 16 64-bit entries), while each of the register banks has a depth of \( N = 16 \) entries. The memory overhead of our stack configuration and corresponding 16 4-bit counters per SRS frame is 16448 bits which is mapped to a single 18Kb BRAM on the FPGA. The area overhead increases linearly with the depth of the banks, i.e., when the depth is doubled, the number of registers is also doubled, and the number of LUTs increases by a factor of 2-3. We also synthesized \texttt{HardScope} using Synopyes Design Compiler targeting the NanGate 45nm Open Cell Library\(^{15}\) which gave a logic size of 200 kGE (Gate Equivalent), equivalent to 800k transistors. The Pulpino SoC utilizes 15444 slice LUTs and 9758 slice registers on the Zedboard which is equivalent to 500 kGE. The area of both the Pulpino SoC and \texttt{HardScope} is very small. To put this in perspective, consider the hardware implementation of a 128-bit high-throughput pipelined AES cipher.\(^{16}\) Such a circuit consumes 12475 slice LUTs and 10769 slice registers on a Virtex-6, on par with \texttt{HardScope}. The reasonable size of \texttt{HardScope} makes it suitable for deployment on a wide variety of SoCs, including MCUs like the Pulpino. If deployed on a modern general-purpose SoC, such as the Apple A10 quad-core ARM64 mobile SoC (roughly 3,300,000,000 transistors), the area overhead of \texttt{HardScope} would be negligible in comparison.

### 7 EXTENSIONS

**On execution context and scope granularity.** RSE could also be used with other granularities of execution context, i.e., either coarser or finer than function granularity. In C and C++, functions are naturally split into smaller code blocks. In Section 5.2 we showed that the function prologue and epilogue can be treated as separate execution contexts from the function body. This is possible, because \texttt{HardScope} instructions have been designed to be agnostic of programming language and language control structures.

For example, consider the simple loop in Listing 5. The \texttt{for} loop forms a separate code block in terms of lexical scope. Since the index variable \( i \) is declared in the loop signature, it cannot be accessed from outside the \texttt{for} loop. The statement in the loop body accesses the \texttt{name} array, the

\(^{15}\)http://www.nangate.com/

\(^{16}\)https://opencores.org/project,aes-128_pipelined_encryption
1|| \texttt{static char name[40];}  
2|| \texttt{static char password[16];}  
3|| void fun(char *buf, int len) {  
4|| \hspace{1em} int i = len - 1;  
5|| \hspace{1em} for (int i = 0; i< len ; i++, x--) {  
6|| \hspace{2em} name[i] = buf[x]; \hspace{1em} // overflows when len > 40  
7|| \hspace{1em} }  
8|| \hspace{1em} // do something with password  
9|| }  

**Listing 5:** C program with a for loop containing a potentially vulnerable memory error. There is no bounds checking when writing to the name buffer.

buf array, and the len integer. The loop can be isolated in a separate execution context from the rest of the function body by surrounding it with \texttt{sbent} and \texttt{sbxit} instructions. Access to name and the buffer referenced by buf are delegated to the loop’s execution context via \texttt{srdlg}, and access to the variable i, which is not accessed from outside the loop, is delegated with \texttt{srdlg}s. This ensures that the for loop is executed with minimal privileges. Should the value of len exceed the size of the name buffer, RSE prevents the buffer operation from overflowing into the password array, which should only be accessed later in the function body.

In addition to execution contexts deduced from C control constructs, like loops, the programmer may use \textit{unnamed blocks} to group related code and data together. Any variables declared inside an unnamed block are considered by the compiler to exist only within the block’s lexical scope. HardScope can make use of this standard language feature to automatically infer developer intent when determining execution contexts during instrumentation.

**Interfacing with legacy code.** The ability to apply RSE instrumentation at varying granularities allows applying coarse-grained RSE for functions that could not otherwise be instrumented, such as pre-compiled shared libraries. Wrapper functions, such as our \texttt{malloc()} wrapper (see Section 5.2) provide function-granularity isolation in terms of delegated objects and stack frame, and provide coarse-grained, module-level isolation for other library data.

8 RELATED WORK

Various software-only and hardware-assisted memory safety technologies have been proposed and/or deployed (e.g., [1, 3–8, 10, 16–18, 24, 25]). We discuss those that aim to mitigate non-control-data attacks. To the best of our knowledge, RSE is the first scheme aiming to mitigate DOP attacks.

\textit{Probabilistic schemes} aim to randomize the data or its layout at run-time so that unauthorized accesses would have unpredictable results. Data Randomization [5] uses static analysis to partition code into equivalence classes, and then instruments all load/store operations to XOR the data with a class-specific mask. Data Space Randomization [4] randomizes the layout of data in memory. However, probabilistic schemes rely on some secret information (e.g., the XOR mask or randomization secret) which if leaked or inferred by the attacker could undermine the scheme.

\textit{Data-Flow Integrity (DFI)} [6] is a software-only approach for mitigating both control-flow and non-control-data attacks. At compile-time, static analysis constructs a data-flow graph of a program. The code is instrumented to construct a table listing the last instruction to write to each individual variable. At run-time, the table is updated on every write operation, and checked before every read operation against the pre-computed data-flow graph. For improved performance, various \textit{equivalence classes} are defined by having multiple instructions share a single identifier. When applied to all variables, DFI incurs a 104% performance overhead, and a 50% memory overhead.

Many approaches operate by associating a base address and upper bound with each pointer, and checking these when the pointer is dereferenced. We denote these schemes as being \textit{pointer-oriented}. \textit{Fat-pointer} schemes add bounds metadata directly to the pointer, e.g., by increasing the length of the pointer [22] or borrowing unused bits from the pointer [16]. However, this changes the memory layout of the program and incurs memory overhead. SoftBound [20] stores the pointer bounds metadata in a disjoint area of memory. Although this retains the program’s original memory layout, it breaks cache locality and leads to additional cache misses when retrieving pointer bounds, and has performance overhead of 67%.

One of the first hardware-assisted techniques for ensuring memory safety was \textit{HardBound} [8]. It introduced the notion of a \textit{hardware bounded pointer primitive}, for which the associated pointer bounds are checked implicitly by the hardware when the pointer is dereferenced. Unlike its software-only predecessors, HardBound stores pointer bounds metadata in a disjoint shadows space, thus retaining the original memory layout. HardBound incurs an average performance overhead of \(\approx 10\%\) and a worst-case memory overhead of \(\approx 200\%\) [8]. However, since HardBound has only been simulated at the micro-operation level, no hardware area overhead or gate delay measurements are available.

Within the SAFE project\footnote{\url{http://www.crash-safe.org/}}, Kwon et al. [18] have presented BIMA, a hardware-assisted fat-pointer scheme. BIMA targets the \textit{simplified SAFE processor}, a clean-slate design which includes various security enhancements (e.g., garbage-collected memory). BIMA encodes pointer bounds metadata within the pointer itself by borrowing unused bits (e.g., on a 64 bit system, it assumes that 46 bit addresses are sufficient). On the SAFE processors, this scheme has no performance overhead and worst-case 16% memory overhead. However, the compact encoding of pointer metadata results in alignment restrictions on pointers. This necessitates the use of custom stack allocators when applying this approach to stack data structures [9].
Intel’s Memory Protection Extensions\(^{18}\) (MPX), introduced in the Intel Skylake microarchitecture in late 2015, is currently the only example of this type of hardware-assisted technology being deployed in real systems. MPX is a pointer-oriented scheme that adds four new registers for pointer bounds and a new instruction for performing the bounds check before the pointer is dereferenced. However, since there are only four bounds registers, most of the pointer-bounds metadata is stored in a special memory region and loaded on demand. Oleksenko et al. \cite{23} found that MPX incurs an average performance overhead of 50% and a memory overhead of 1.9x, largely due to the time and memory required for storing and loading bounds metadata.

Software Fault Isolation (SFI) \cite{7, 10, 27} mitigates the consequences of both non-control-data and control-flow exploits by isolating software components, such as kernel modules and dynamically loaded libraries, into distinct protection domains.

Run-time attestation, such as, Control-Flow Attestation \cite{2}, can detect both control-flow and some non-control-data attacks, but are unsuitable as preventive measures.

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 \cite{6} and Soft-Bound \cite{20}), HardScope has significantly lower overhead, does not require whole-program static analysis, and can enforce different rules during different invocations of the same function (context-specific memory isolation). HardScope does not require any additional input from developers (cf. YARRA \cite{24}).

Unlike previous hardware schemes, HardScope is block-oriented as it considers blocks of code, rather than individual pointers (pointer-oriented), as the subjects of the access control rules. This eliminates the need to load or calculate different sets of bounds when using multiple pointers within the same code block (cf. MPX \cite{23}). It also reduces the amount of metadata that must be stored when a single code block contains multiple pointers that access the same data, thus making it feasible to store this metadata in on-chip memory. HardScope retains the program’s original memory layout and does not require any changes such as redzones (cf. AddressSanitizer \cite{25}) or special alignment of pointers (cf. low-fat pointers \cite{9, 18}).

9 CONCLUSION
The prevalence of data-oriented gadgets in real-world software and the expressiveness of DOP attacks makes these attacks challenging to mitigate. Multiple DOP attacks have been demonstrated, but no suitable mitigation techniques are yet available. We propose run-time scope enforcement (RSE), a novel approach to protect memory at run-time using lexical scope information about variables already available at compile-time. We present HardScope, our proof-of-concept implementation of hardware-assisted RSE targeting the RISC-V instruction set architecture. HardScope introduces a new set of instructions that can be automatically inserted by a compiler to protect run-time data from out-of-scope accesses. We show how the context-specific inter-procedural memory isolation provided by HardScope mitigates all currently known DOP attacks at multiple points in each attack. Our synthesis of HardScope shows that it can be realized while incurring reasonable area overhead on the small RISC-V Pulpino SoC, and our cycle-accurate simulations demonstrate a performance overhead of 7.1% when providing complete mediation of all memory accesses. HardScope can also enforce memory isolation at coarser or finer granularity, to facilitate different types of memory protection strategies. We provide a version of the official RISC-V simulator, with added support for HardScope instructions, to support reproducibility of our results.

ACKNOWLEDGMENTS
This work was supported by the German Science Foundation CRC 1119 CROSSING project S2, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU’s Horizon 2020 research and innovation program under grant number 643964 (SUPERCLOUD), Tekes — the Finnish Funding Agency for Innovation (CloSer project), and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

\(^{18}\)https://software.intel.com/en-us/isa-extensions/intel-mpx
REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. ACM Trans. Inf. Syst. Secur. 13, 1, Article 5 (Nov. 2009), 40 pages. DOI: https://doi.org/10.1145/1609556.1609960

[2] Tigest Abera, N. Asokan, Lucas Davi, Jan-Erik Elberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsjik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16). ACM, New York, NY, USA, 743–754. DOI:https://doi.org/10.1145/2976749.2976838

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP ’08). IEEE Computer Society, Washington, DC, USA, 263–277. DOI:https://doi.org/10.1109/SP.2008.30

[4] Sandeep Bhattachar and R. Sekar. 2008. Data Space Randomization. In Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’08). Springer-Verlag, Berlin, Heidelberg, 1–22. DOI: https://doi.org/10.1007/978-3-540-70542-0_1

[5] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. 2008. Data Randomization. Technical Report. 14 pages. https://www.microsoft.com/en-us/research/publication/data-randomization/

[6] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI ’06). USENIX Association, Berkeley, CA, USA, 147–160. http://dl.acm.org/citation.cfm?id=1298455.1298470

[7] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP ’09). ACM, New York, NY, USA, 45–58. DOI:https://doi.org/10.1145/1629575.1629581

[8] Joe Deviett, Colin Blundell, Milo K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII). ACM, New York, NY, USA, 103–114. DOI: https://doi.org/10.1145/1346281.1346295

[9] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. (2017).

[10] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. Stack Bounds Protection with Low Fat Pointers. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI ’06). USENIX Association, Berkeley, CA, USA, 75–88. http://dl.acm.org/citation.cfm?id=1298455.1298463

[11] Chris Evans. 2016. Advancing exploitation: a scriptless 0day exploit against Linux desktops. (2016). https://scarybeastsecurity.blogs.com/2016/11/0day-exploit-advancing-exploitation.html

[12] Eclipse IoT Working Group, IEEE, Agile-IoT EU, and IoT Council. 2017. IoT Developer Survey 2017. (2017). https://janskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/

[13] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagen德拉 Modadugu, and Dan Boneh. 2004. AddressSanitizer: A Fast Address Sanity Checker. In Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/addresssanitizer

[14] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS ’13). ACM, New York, NY, USA, 721–732. DOI:https://doi.org/10.1145/2508859.2516713

[15] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP ’93). ACM, New York, NY, USA, 203–216. DOI:https://doi.org/10.1145/168619.168635

[16] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14). USENIX Association, Berkeley, CA, USA, 147–163. http://dl.acm.org/citation.cfm?doid=2685048.2685061

[17] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP ’93). ACM, New York, NY, USA, 203–216. DOI:https://doi.org/10.1145/168619.168635
A EXCERPT FROM SREPLACE

Listing 6 shows an excerpt from the vulnerable `sreplace()` function in ProFTPD.

```c
1   char *sreplace(pool *p, char *s, ...) {
2       char *src = s; // Pointer to next input character
3       char *cp; // Pointer to next free entry in buf
4       char **mptr; // Pointer to replacement pattern
5       char **rptr; // Pointer to replacement string
6       char *marr[33]; // Replacement pattern string array
7       char *rarr[33]; // Replacement string array
8       *pbuf = NULL; // Pointer to beginning of buf
9       //
10      /* out buffer */
11     char buf[PR_TUNABLE_PATH_MAX] = {'\0'};
12     // unsigned sizes of replacement pattern, string and buf
13     size_t mlen = 0, rlen = 0, blen;
14     cp = pbuf = buf;
15     blen = sizeof(buf);
16     while (*src) { /* while input not exhausted... */
17         // ...match substring against all replacement patterns
18         for (marr = marr, rptr = rarr;
19             *mptr; // null indicates end of array
20             mptr++, rptr++) {
21             mlen = strlen(*mptr); // length of pattern
22             rlen = strlen(*rptr); // length of string
23             if (strncmp(src, *mptr, mlen) == 0) {
24                 strncpyp(cp, *rptr, mlen); // copy replacement string to buffer
25                 cp = cp + mlen;
26                 // copy character verbatim
27                 *rptr = *mptr;
28                 // destination string
29             } else if (!*mptr) { /* no replacement pattern matched */
30                 // Off-by-one bounds check
31                 if ((cp - pbuf + 1) > blen) {
32                     cp = pbuf + blen - 1;
33                     // copy character verbatim
34                     *cp++ = *src++;
35                 }
36             }
37         }
38     }
39     return pbuf;
40 } /* sreplace */
```

Listing 7: Excerpt of ProFTPD’s `pr_strtime()` function with `mons` array. `pr_strtime()` indexes `mons` by month number, copies the corresponding string literal to the static output buffer, and returns a pointer to the output buffer.

C OTHER KNOWN DOP ATTACKS

C.1 DOP attacks against ProFTPD

In addition to the DOP program that bypasses randomization defenses discussed in Section 6.1, Hu et al. [14] present two additional end-to-end attacks against ProFTPD. Both attacks build on the idea of leveraging DOP to corrupt relocation information maintained by the Linux runtime linker. The runtime linker maintains a `link_map` structure for each loaded ELF object. The `link_map` structure contains the name of the corresponding ELF object, the base address at which the object is loaded, and the virtual address of all the ELF object’s dynamic metadata tables.

In the DOP attack, the memory corruption vulnerability in `sreplace()` is used to inject specially crafted symbol and relocation metadata into the program’s data section. The linked list of `link_map` structures is then corrupted to include the malicious relocation metadata. The malicious relocation metadata is consumed by the POSIX `dlopen()` function. `dlopen()` is responsible for patching the relocated addresses before execution of the loaded module. In order to do so, `dlopen()` has the ability to modify arbitrary memory locations, even code pages or read-only data sections. ProFTPD invokes `dlopen()` in its PAM module to dynamically load libraries. By corrupting a field in a global static data structure used by PAM subsystem to keep track of loaded modules, the attacker can trigger the invocation of `dlopen()`. As `dlopen()` is invoked, it will process the malicious relocation metadata and trigger an arbitrary memory corruption. This gives the attacker a powerful DOP gadget capable of bypassing defenses based on non-writable code or read-only data.

RSE prevents the attacker from escalating the `sreplace()` assignment gadget into a powerful `dlopen()`-based assignment in two ways. First, RSE can prevent `sreplace()` from tampering with the `link_map` structure. Second, it can prevent `sreplace()` from modifying the data structures belonging to the PAM subsystem.

B EXCERPT FROM PR_STRTIME

Listing 7 shows an excerpt from the `pr_strtime()` function in ProFTPD [14].
C.2 DOP attacks against Wireshark

The attacker must corrupt the loop condition `cell_list` in the parent function `gtk_tree_view_column_cell_set_cell_data()` to ensure that the loop executes infinitely. To achieve this, the `cell_list` is directed to a malicious linked-list in the malicious payload.

RSE prevents the initial buffer overflow by ensuring that `packet_list_dissect_and_cache_record()` function is only delegating access the `pd` array. Thus `cf_read_frame_r()` is prevented from accessing variables in its caller's stack frame.

Similarly, ensuring the availability of the gadget dispatcher requires an out-of-scope access from `packet_list_change_record()` to the `cell_list` variable local to `gtk_tree_view_column_cell_set_cell_data()`. RSE can thus prevent the DOP program establishing a reliable gadget dispatcher for the attack.

C.3 DOP attacks against GStreamer

The attacker control the contents of the heap object. Thus `cf_read_frame_r()` is reachable by corrupting the `flxdec->frame_data` array. Thus `cf_read_frame_r()` is prevented from accessing variables in its caller's stack frame.

In order to chain gadgets found in `packet_list_dissect_and_cache_record()` and `packet_list_change_record()`, the attacker must corrupt the loop condition `cell_list` in the parent function `gtk_tree_view_column_cell_set_cell_data()` to ensure that the loop executes infinitely. To achieve this, the `cell_list` is directed to a malicious linked-list in the malicious payload.
struct _GstFlxDec {
    GstElement element;
    GstPad *sinkpad;
    GstPad *srcpad; // pointer to gst_flxdec_chain()
    gboolean active, new_meta;
    guint8 *delta_data, *frame_data;
    GstAdapter *adapter;
    guint size;
    GstFlxDecState state;
    gint64 duration;
    FlxColorSpaceConverter *converter;
    FlxHeader hdr;
    FlxHeader hdr;
};

Listing 10: Excerpt from gst-plugins-good/gst/flx/gstflxdec.h [11].

Listing 11: Excerpt from gst_flxdec_chain() in
gst-plugins-good/gst/flx/gstflxdec.c [11]. The data-oriented gadgets Θ and Θ are reachable by corrupting the flxdec heap object.

Evans [11] demonstrates an attack against the GSTreamer FLIC decoder\(^{21}\) that exploits a combination of control-flow hijacking and DOP techniques. The attack exploits a decode loop that lacks bounds checks against the output frame_data buffer, as shown in Listing 9. The exploit can be triggered via a specially crafted media file that causes the decode loop to write past the bounds of the heap-resident buffer. The goal of the attacker is to escalate the buffer overflow into an arbitrary code execution, but they cannot perform a traditional control-flow attack due to the presence of ASLR. The initial memory corruption vulnerability is non-linear, allowing the attacker to skip over heap memory before the write, but only allows the attacker to tamper with memory below the overflowing frame_data buffer. Additionally the attacker targets program logic that extracts metadata from a media file which only runs the decode loop for two media frames, considerably limiting the initial number of non-linear forward write gadgets possible to chain together.

GStreamer attack operates by corrupting a heap object which is legitimately within the scope of the decode loop with the initial memory corruption vulnerability. Thus, preventing the GStreamer attack is more challenging compared to the ProFTPD attacks described in Section 6.2 and Appendix C.1. However, RSE can be utilized by the caller to selectively delegate individual fields of the dynamically allocated flxdec in flx_decode_delta_fli() gst_flxdec_chain. Furthermore, by applying RSE at a block-granularity, the vulnerable decode loop can be constrained to stay within the bounds of the flxdec->delta_data and flxdec->frame_data buffers.

\(^{21}\)https://gstreamer.freedesktop.org
D PERFORMANCE OPTIMIZATION

As discussed in Section 5.3, HardScope context switch instructions `sbent` and `sbxit` require at most $N$ additional cycles to transfer the topmost frame between stack and cache in both directions. This implies that the processor must stall if another context switching instruction is encountered within these $N$ cycles. One way to improve the performance at no additional area cost, is to overclock the stack at a frequency higher than that of the processor, i.e., if the processor’s operation frequency is $x$ then the stack operation frequency can be set at $2^n x$, where $1 < n \leq \log_2 N$. This would effectively reduce the number of stall cycles to $N/2^n$. Note that this comes at the cost of increased power consumption and might not be feasible if the processor is already operating at a high clock frequency.