ACES - RISCV: Automatic Compartments for Embedded Systems on RISCV

CS7999 Report

Computer Science & Engineering Department
IIT MADRAS

May 2022

CS20D408
Sai Venkata Krishnan V

Contents

1 Introduction

ACES[1] presents an automated compiler setup to infer and enforce automatic data and code compartmentalization in baremetal embedded systems. This is based on the principle of least privileges, which is an important cyber security practice, aimed at preventing both known and 0-day attacks. Defense against 0-day attacks is crucial in embedded systems, as most of them are deployed in varying geographies were remotely patching an existing software with latest vulnerability mitigation code might be difficult and also would depend on the security practices followed by the end-client owning the device. ACES splits the final executable into a set of code regions along with its set of data and memory-mapped peripheral compartments. ACES makes use of existing hardware feature called ARM Memory Protection Unit(MPU) to restrict and isolate the data and instruction accesses that can be performed by a code compartment.

Currently ACES supports only the ARM backend. We have ported the ACES design to RISCV backend, where instead of ARM MPU, we utilize RISCV Physical Memory Protection(PMP) to restrict code and peripheral accesses to within a compartment. Our current RISCV port of ACES does not implement data compartments.

The rest of the report is organized as follows: In Section 2 we briefly discuss existing compartmentalization techniques in embedded systems, ARM MPU and RISCV PMP features. In Section 3 we discuss the original ACES design and its key features. In Section 4 we discuss our implementation of ACES on RISCV.

2 Background

2.1 Principle of Least Privileges

The Principle of least privileges is a well-known cyber security concept that aims to protect the system against unknown attacks. Even if an attacker is able to gain entry into a compartment, the attacker is restricted only to that compartment, and the vulnerability is confined and prevented from spreading to the rest of the system.

2.2 Compartmentalization Techniques in Embedded Systems

Mbed μvisor[2] is an ARM based compartmentalization feature, which allows the programmer to annotate the source files and create compartments within baremetal applications. It restricts the data and peripheral accesses to subsets of code, but the code region as such is fully accessible. MINION[3] is another compartmentalization based solution for embedded systems which automatically infers and enforces thread level compartment policies. Both these solutions are not widely adopted due to two main reasons: (i) Coarse grained compartmentalization policies are only allowed which leads to fewer security guarantees. (ii). Compartmentalization policy is either fixed irrespective of the application it is applied to or is required to be fully specified by the program developer itself by annotating the source code. This significantly inhibits the adoption of compartmentalization solutions. An ideal compartmentalization solution should be flexible and aid the program developers in choosing the right policy for each application. Moreover, it should be easy for the developer to try out various compartmentalization policies for an application without changing its implementation,

2.3 ARM Memory Protection Unit(MPU)

ARM MPU enables restricting accesses to physical memory through a set of registers. Based on the underlying ARM processor the number of MPU registers varies and is fixed. Each MPU register points to a contiguous memory region and sets the RWX permission to allow/restrict access accordingly. Each MPU register identified memory region should satisfy some alignment constraints for efficient implementation. Two or more MPU regions can overlap, but the access permissions set by the higher numbered MPU register is given precedence.

2.4 RISCV Physical Memory Protection(PMP)

RISCV PMP[4] is similar to ARM MPU in their ability to protect physical memory regions. The number of PMP registers are fixed and varies from each processor. The standard configuration is 16 registers. To enforce PMP, two sets of CSRs are used: *pmpaddr* and *pmpcfg*. The *pmpaddr* denotes the address range and the *pmpcfg* denotes the permissions and alignment constraints on the address range stored in *pmpaddr*. PMP is effectively used to protect M-mode (machine mode) only accessible memory from S-mode(supervisor mode) or U-mode(user mode). The PMP related CSRs can be modified from the M-mode code but not from S or U-mode code. This helps in preventing the attacker from maliciously changing the PMP CSRs to allow unconstrained access to memory region. A special bit called *Lock* bit in *pmpcfg* CSR if set, disallows even the M-mode to change that specific PMP CSR entry once it is set. Similar to ARM MPU, priority is given to lower numbered PMP registers while checking for Address Matching.

3 ACES Design

3.1 Threat Model

1. Attacker can perform exploit any memory vulnerability
2. System is running a single baremetal statically linked application.
3. Source code should be available.
4. Software is trustworthy but might contain bugs (not malicious).
5. The underlying hardware is trusted.

3.2 ACES: Automatic Compartments for Embedded Systems

ACES seeks to automatically infer the optimal code and data compartmentalization policy given an application's source code and maps those compartments to the MPU registers to restrict the code/data accesses of the compartment at runtime. This is achieved in the following way:

1. Program dependence graph(PDG) of the source file is constructed and various types of nodes are identified: global variables, peripherals, functions.
2. Graph partitioning algorithm is applied to PDG to form an Initial Region(Compartment) Graph, where each function is placed in its own compartment and edges from each compartment node to all the data and peripheral nodes that it accesses are added to the graph.

3. User defined compartment policy is fed to initial region graph to group functions based on compartment policy. By grouping, even the associated data and peripheral edges are also merged in the resulting compartmentalized region graph

4. The compartmentalized region graph is modified based on any developer optimizations that might be introduced to improve performance or security.

5. The optimized compartment region graph is lowered to satisfy MPU constraints. By lowering, multiple compartments are merged. The lowering process makes use of cost functions that aim to minimize the overall data and peripheral accesses allowed for each merged compartment thereby maximizing the isolation of the compartments and increasing security guarantees.

6. To allow cross-compartment control flow transfers, the set of functions that are involved in cross-compartment invocations/returns are identified and all their call and return instructions are replaced with SVC(supervisor call) instructions. On SVC execution, execution traps to a trusted Security Monitor which validates the cross-compartment transfer based on the caller and callee's PC values and identifies whether it is a cross-compartment invocation or not. Registers are saved and restored to prevent information leakage. The security monitor is also responsible for setting up the MPU registers based on the target compartment to which it would pass on execution to.

7. Finally in the linking stage the memory layout of the compartments is set and isolated by configuring the MPU registers that each compartment would be assigned when it is executing.

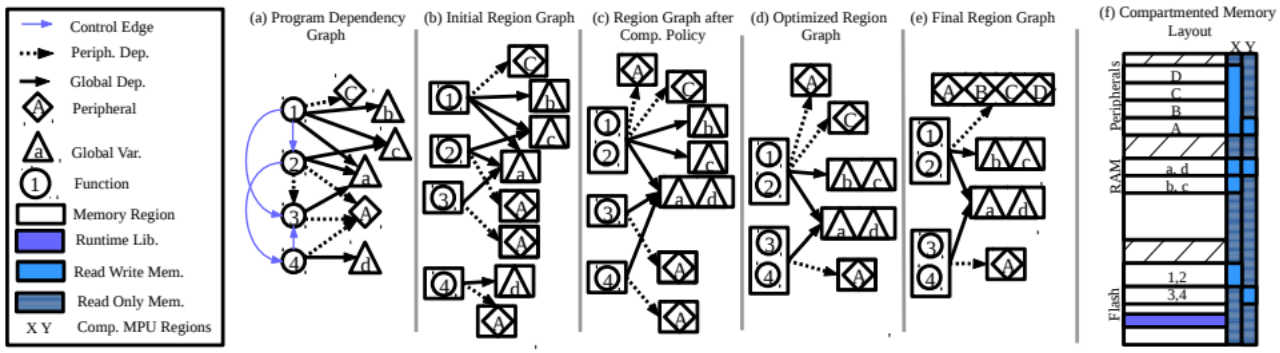

Figure 1: Compartment Creation Process [1]

The above steps are shown in Figure 1. The stack memory region for each compartment is restricted to all addresses that lie below the original stack pointer value at the start of compartment execution. But this is might lead to program crashes since the stack memory contents of regions above the stack pointer might also be accessed. To account for this, a software based Micro-emulator is used to trap those special memory accesses and perform the load/store operation only if the address is whitelisted to be accessed by that compartment. The set of whitelisted addresses for each compartment is determined during the testing phase, where the ACES enabled application is monitored on different inputs.

Since the MPU registers are fixed for a given hardware, the number of data-peripheral regions that each compartment has access to is limited. These constraints are also taken into consideration when the region graph is lowered. Mapping constant-address memory-mapped peripherals to compartments have to obey memory alignment constraints which leads to groupings of closely located(spatially in memory) peripherals.

ACES requires source code to infer and enforce compartment strategies. To facilitate a secure interface with pre-compiled dependent libraries, all the pre-compiled library code region is placed in an always-executable MPU region and set through the application runtime. The data and peripherals those libraries can access is limited to those that can be accessed by the compartment which invokes the library function.

ACES separates the compartment policy from program execution, which makes it more flexible and easily adoptable.

3.3 Key Takeaways

1. Compartment based solutions are crucial in protecting IoT systems where updates are more difficult to perform and constrained. So IoT systems need to be well protected against unknown 0-day attacks.

2. Finding the right compartment strategy varies with application and requires the expertise of both the security engineer as well as software developer, to achieve the optimal performance-security trade-offs.

3. Tools to aid the engineers to find optimal compartment strategies are crucial in making it more adoptable and usable.

4 ACES on RISCV

4.1 Design

Our project aims to port ACES, function compartmentalization in particular to the RISCV backend utilizing RISCV PMP. ACES codebase w.r.t ARM backend is completely open-sourced[5]. LLVM link-time analysis passes where enabled to build the program dependence graph. These analysis passes were tweaked to support LLVM-11 version(ACES ARM backend is implemented on LLVM-4 but RISCV backend is supported from LLVM-11). Following are the design steps to extract function compartments and embed these metadata into the code for runtime checks.

1. **Analysis Phase**: LLVM link time analysis pass analyses the source code at link time, and generates the data and control flow dependencies. To enable LTO analysis, all the source code has to be compiled with *-flto* optimization flag. The dependency results are dumped into JSON file.

2. **Compartment Policy Making Phase**: The dependency analysis results present in the JSON file is passed to graph-analysis python script which tries to merge functions into compartments and assign compartment id based on a compartment policy. This python script is mostly inspired from open-sourced ACES script. Currently 3 policies are supported:

(a) **Peripheral policy**: Compartments are created based on which code regions/functions should have access to a specific peripheral or common set of peripherals.

(b) **Naive Filename policy**: Compartments are created by grouping functions present in the same source code file together. To support this compartment policy, the analysis phase should compile all the instrumented code with *-g* flag so that the file name metadata associated with a function can be retrieved during the link time analysis.

(c) **Optimized Filename policy**: Naive filename policy is fine-grained. Whereas, optimized filename policy tries to merge compartments, which frequently call each other, into single compartment. This policy tries to balance security-performance trade-off issue. The minimum number of inter-compartment edges that should be present inorder to deem that merging two compartments would improve performance should be determined carefully, since merging of compartments increases the available attack surface present in the merged compartment.

Other custom policies can also be implemented. The advantage with ACES, as stated before, is that defining the compartment policy is not tied to the source code.

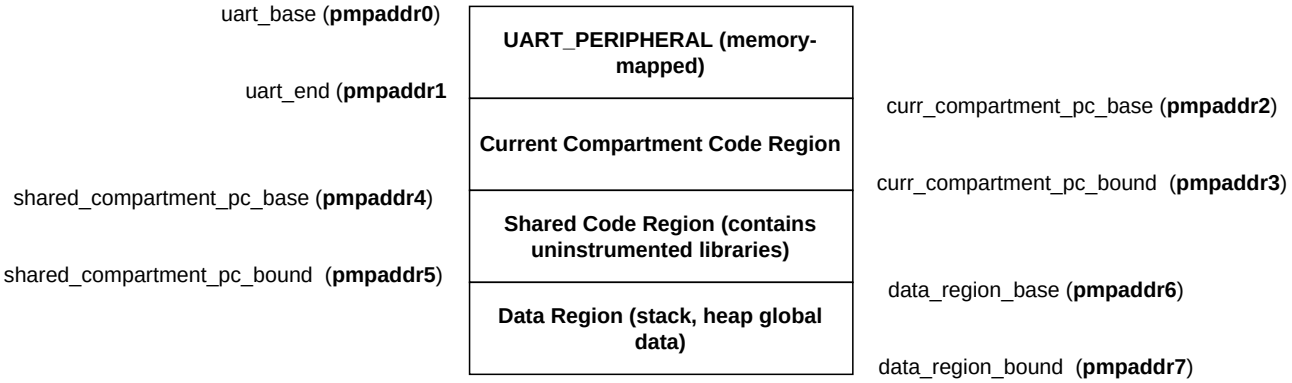

Figure 2: ACES-RISCV PMP Regions

3. **Compartment Policy Enforcement Phase**: This phase consists of two steps:

(a) **Cross-call and Cross-return Tagging**: All call instructions whose target set includes any function which does not belong to the caller's compartment, then that call instruction is specified as a *cross-call* instruction. Within a compartment all *cross-call* instructions are identified and tagged uniquely. Tagging is achieved by populating a temporary register based on RISCV ABI. Similarly all the return instructions present in functions which might be a target of the *cross-call* instructions are also tagged as *cross-return* instructions. The tagging of *cross=return* instruction need not be unique. For indirect *cross-call* instructions, the set of target functions is over-approximated to the set of all *address-taken* functions which have a function type similar to the function type targeted by the *cross-call* instruction. Each function which contains a *cross-return* instructions is placed in a new sub-section belonging to their parent-compartment's section.

(b) **Custom Linker Script and Startup Code Generation**: A custom linker script is generated based on the cross-call-target list and the cross-return functions. At runtime the list of valid targets that a cross-call can jump to is fetched and checked to restrict control-flow. This data is populated with help of linker script at startup time of code by exporting the symbol references of the starting and ending addresses of each compartment and each *cross-return* function.

4. **Startup Phase**: At startup, all metadata are initialized. Metadata includes the PC base and bound of each compartment, valid target list of each *cross-call* instruction in each compartment, secure stack used during the compartment switch handler. All these metadata and stacks are stored in a separate data section which does not have corresponding PMP entry, so as to prevent the attacker from modifying these values at runtime. The bootup code also initializes the PMP registers. Totally 8 PMP registers are used, to denote 4 regions as shown in Figure 2. The corresponding *pmpconfig* registers, as shown in Figure 3 are also set with RWX permission based on whether they represent a code or data region. Once the bootup code is finished, the mode is switched to User mode and application code is executed. This is crucial because, running the application code which is vulnerable in M-mode can be exploited by the attacker to access and change PMP registers, since by default M-mode can access all memory regions and PMP registers.

pmp0cfg	0
pmp1cfg	TOR | R | W
pmp2cfg	0
pmp3cfg	TOR | R | X
pmp4cfg	0
pmp5cfg	TOR | R | X
pmp6cfg	0
pmp7cfg	TOR | R | W

Figure 3: ACES-RISCV *pmpcfg* configuration

Algorithm 1 ACES-RISCV Compartment Switch Handler

Require: Input $a = CSR\_MEPC$ ▷ MEPC CSR: callee's PC
Require: Input $b = REG\_RA$ ▷ Return address register: caller's PC
Require: Input $c = CSR\_MCAUSE$ ▷ MCAUSE CSR
Require: Input $d = REG\_X31$ ▷ Unique ID for cross-call within compartment
Require: Input $e = REG\_X30$ ▷ Tail call detection

procedure HEXBOX_HANDLER(a, b, c, d, e)
  if $c \neq 1$ **then** ▷ Code compartments only supported
    $generic\_trap\_handler(c, a)$
  end if
  if $d \neq 0$ **then** ▷ Compartment entry handler
    $c1 \leftarrow get\_compartment\_id(b)$
    $c2 \leftarrow get\_compartment\_id(a)$
    $target\_list \leftarrow get\_target\_list(c1, d)$
    if a *in* $target\_list$ **then**
      $target\_pc\_bounds \leftarrow get\_pc\_base\_bound(c2)$
      $update\_pmp\_registers(target\_pc\_bounds)$
      if $e == 0$ **then**
        $hexbox\_stack[top++] \leftarrow b$
        $clear\_temporary\_registers()$
      end if
    end if
  else if $d == 0$ **then** ▷ Compartment exit handler
    if $hexbox\_stack[top]$! =a **then**
      $ERROR(\text{"}Invalid\ Control\ Flow\text{"})$
    end if
    $c1 \leftarrow get\_compartment\_id(b)$
    $target\_pc\_bounds \leftarrow get\_pc\_base\_bound(c1)$
    $update\_pmp\_registers(target\_pc\_bounds)$
    $clear\_temporary\_registers()$
  end if
end procedure

5. **Runtime Enforcement**: At runtime, whenever an instruction access fault, occurs we assume it to be because of RISCV PMP access violation. The access fault is trapped by the *HEXBOX_HANDLER*1 which retrieves the *cross-call*/*cross-return* instruction's PC, the destination PC value, and the tag of the caller's instruction stored in a temporary register. Based on the tag, whether the access-violation originated due to a cross-call instruction or cross-return instruction can be figured out. If the tag is:

   (a) **Cross-call instruction**: The list of targets associated with that specific cross-call instruction belonging to that compartment is retrieved and checked with the callee's PC value. If it matches, then the compartment switch passes and the PMP registers are changed. Also, the return address is pushed into a secure(PMP-protected) stack. If the call instruction is a tail-call then the return address is not pushed onto the secure stack.

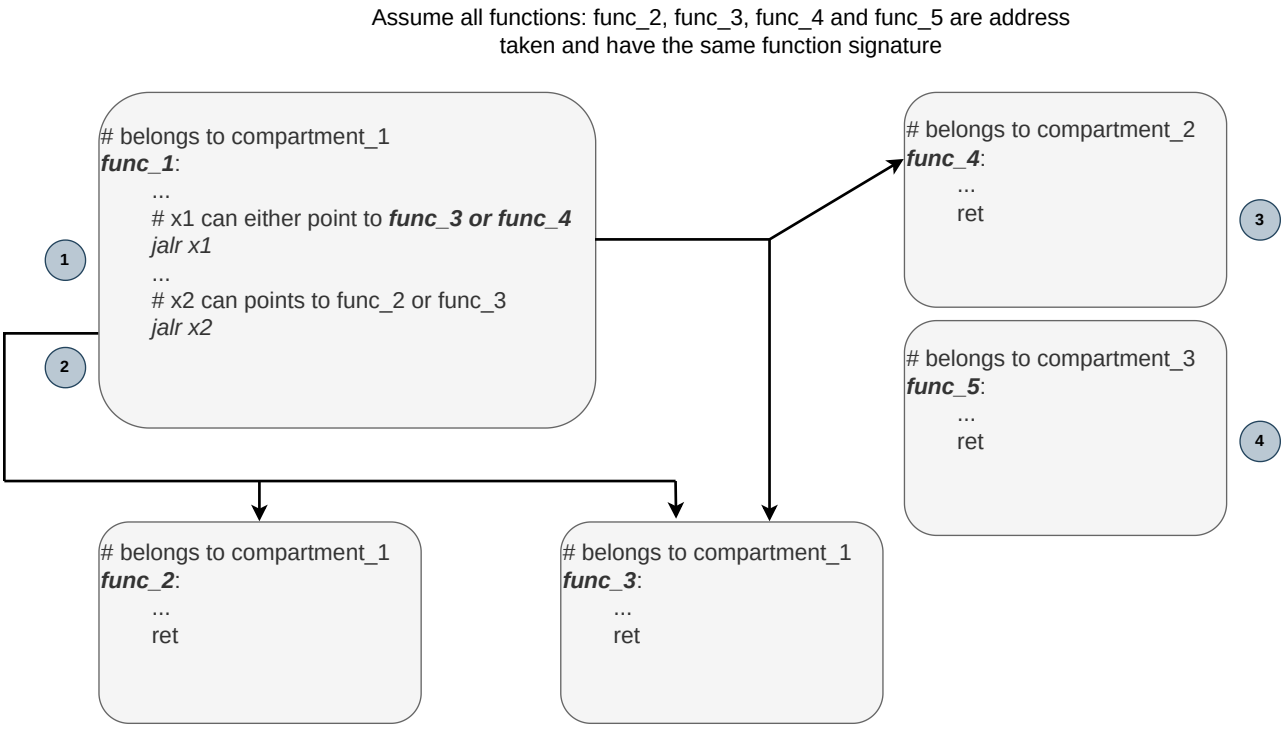

Figure 4: ACES-RISCV Analysis Phase

(b) **Cross-return instruction**: If the caller's tag corresponds to a *cross-return* instruction, then the callee's value is checked with the return address stored at the top of the secure stack. If it matches then the compartment exit is handled by populating the PMP registers with the appropriate values w.r.t target compartment. The secure stack methodology is similar to that of the shadow stack, the only difference being that this shadow/secure stack is populated only when cross-compartment calls/returns occur, i.e. when the target PC value of the control flow instruction fails to fall within any of the active PMP register entries.

An example instrumentation of *cross-call* and *cross-return* instruction is shown in Figures 4 and 5. In Figure 4, it is identified that 1st jump instruction in *func_1* is a *cross-call* instruction and the second instruction is intra-compartment call instruction based on indirect target analysis. Now, all the functions having the same function type as that of cross-call instruction are valid targets. So even though *func_5* would not be called by *func_1*, it is still added to list of targets due to matching function signature. All *cross-call* and *cross-return* instruction are modified as shown in Figure 5.

4.2 Differences between ACES-ARM and ACES RISCV Implementation

Our code instrumentation of of ACES on RISCV slightly differs from that of ARM.

1. In ACES-ARM, the target list for each *cross-call* instruction is inlined within the code section adjacent to *cross-call* instruction itself. In our ACES-RISCV, we store all those data

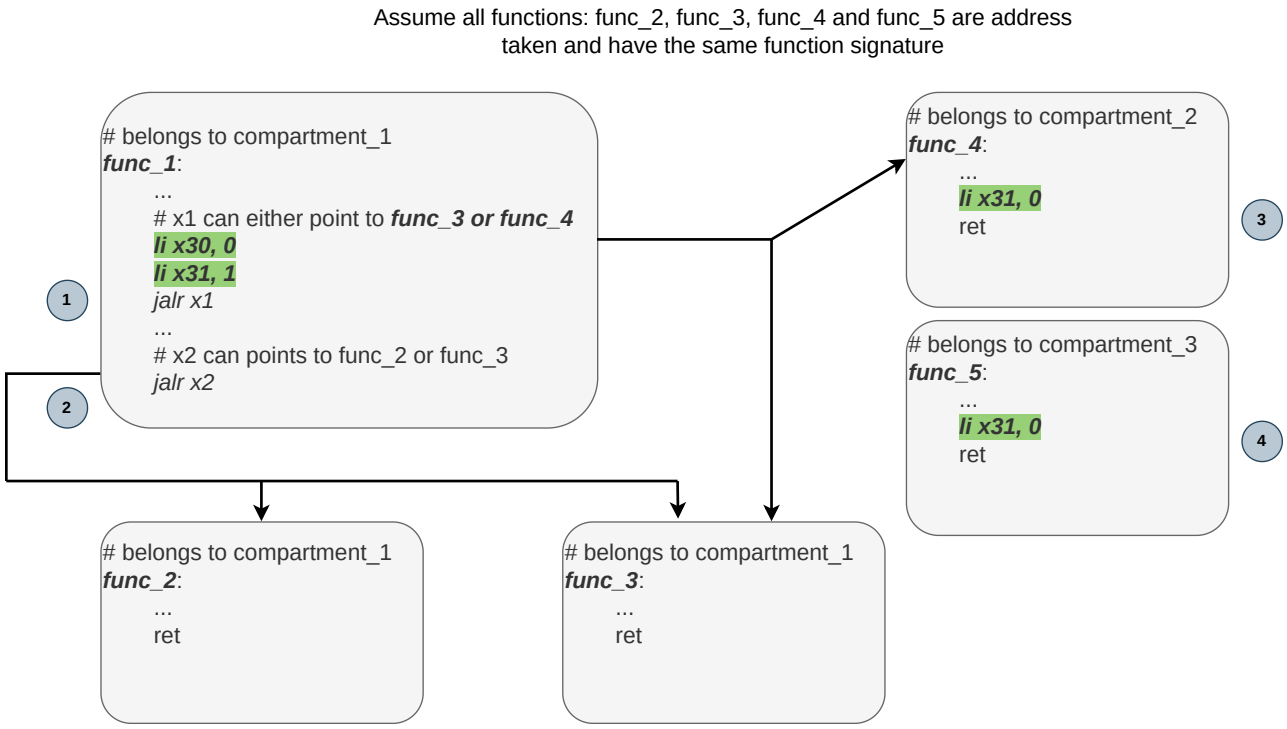

Figure 5: ACES-RISCV Instrumentation Phase

separately in a separate section protect by PMP entry.

2. To identify *cross-call* and *cross-return* instruction, ACES-ARM replaces all the *cross-call* and *cross-return* instructions with a $SVC < code >$ supervisor call instruction which traps to the compartment switch handler. The *code* immediate field of SVC instruction is set to either 100 or 101 based on whether it is a cross-call or cross-return instruction. In this approach a MPU access fault exception does not originate, instead a custom exception w.r.t compartment entry and exit is generated before the target PC is even executed. Also, the need for uniquely identifying each *cross-call* instruction is not required as the metadata is inline adjacent to the *cross-call* instruction itself.

   In our ACES-RISCV approach, instead of raising a exception on each *cross-call* or *cross-return* instruction, we rely on PMP access failure exception when the out-of-bound target PC is executed. We then handle the exception by looking at the unique id of the *cross-call* instruction and perform runtime checks. The necessity for the unique-ids is because we are maintaining the metadata separately and not inlining along with the *cross-call* instruction.

3. Not every instance of a *cross-call* or *cross-return* instruction actually switched compartments, i.e. targets a out-of-bound instruction. It might call functions within the same compartment also. For example in Figure 4, the *cross-call* instruction can call either *func_3*/*func_4*/*func_5*. Out of these 3, only the *func_4* and *func_5* are in different compartments. So the compartment switch exception need not be raised at all when the runtime target of the *cross-call*/*cross-return* instruction is within the same compartment. ACES-ARM's design choice of replacing

cross-call/*cross-return* with *SVC* instructions leads to an exception being raised even when the target PC is within compartment. This increases the performance overhead. Whereas, ACES-RISCV's design of throwing exception only when the out-of-bound instruction is executed has lesser overhead as it optimizes the intra-compartment *cross-calls* and *cross-returns*.

4. ACES-ARM requires the compartment sections to be memory aligned so as to pass the ARM MPU constraints, which results in memory fragmentation and wastage. ACES-RISCV currently does not impose such memory alignment constraints, as RISCV PMP supports contiguous memory ranges with 4-byte alignment constraints. But the downside is that the number of PMP registers used to restrict entry to a memory region increases. This would become an issue when data compartments are also enabled, which restrict the number of active memory regions that can be accessed by the currently executing code compartment. In such a case, switching aligned mode of RISCV PMP would e beneficial.

5 Future work

1. Profiling information and taint analysis information can be incorporated to arrive at better compartmentalization policies. For example, instructions dealing with tainted data can be deemed vulnerable and placed in a different compartment.

2. Source code might not be available always. It would be interesting to see how binary analysis and rewriting techniques can be applied to achieve code compartments.

3. Inlining of metadata in code-region can be performed to optimize the lookup time of the compartment switch handler while retrieving the *cross-call* target list.

4. We have not supported data-compartmentalization in ACES-RISCV. currently. It is not clear for how long the recording/testing phase is enabled to generate a whitelist of address and what guarantee it provides that all benign addresses that could be accessed were recorded.

References

[1] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 65–82. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/clements

[2] ZDNet. mbed os. [Online]. Available: https://www.mbed.com/en/development/mbed-os/

[3] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," 01 2018.

[4] RISCV-Foundation. Riscv privileged spec. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[5] ACES. Aces github repository. [Online]. Available: https://github.com/embedded-sec/ACES/