ACES: Automatic Compartments for Embedded Systems

Abraham A. Clements
Purdue University and Sandia National Labs
clemen19@purdue.edu

Naif Saleh Almakhdhub
Purdue University
nalmakhd@purdue.edu

Saurabh Bagchi
Purdue University
sbagchi@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Abstract

Securing the rapidly expanding Internet of Things (IoT) is critical. Many of these "things" are vulnerable bare-metal embedded systems where the application executes directly on hardware without an operating system. Unfortunately, the integrity of current systems may be compromised by a single vulnerability, as recently shown by Google's P0 team against Broadcom's WiFi SoC.

We present ACES (**A**utomatic **C**ompartments for **E**mbedded **S**ystems)[1], an LLVM-based compiler that automatically infers and enforces inter-component isolation on bare-metal systems, thus applying the principle of least privileges. ACES takes a developer-specified compartmentalization policy and then automatically creates an instrumented binary that isolates compartments at runtime, while handling the hardware limitations of bare-metal embedded devices. We demonstrate ACES' ability to implement arbitrary compartmentalization policies by implementing three policies and comparing the compartment isolation, runtime overhead, and memory overhead. Our results show that ACES' compartments can have low runtime overheads (13% on our largest test application), while using 59% less Flash, and 84% less RAM than the Mbed μVisor—the current state-of-the-art compartmentalization technique for bare-metal systems. ACES ' compartments protect the integrity of privileged data, provide control-flow integrity between compartments, and reduce exposure to ROP attacks by 94.3% compared to μVisor.

1 Introduction

The proliferation of the Internet of Things (IoT) is bringing new levels of connectivity and automation to embedded systems. This connectivity has great potential to improve our lives. However, it exposes embedded systems to network-based attacks on an unprecedented scale. Attacks against IoT devices have already unleashed massive Denial of Service attacks [30], invalidated traffic tickets [14], taken control of vehicles [23], and facilitated robbing hotel rooms [8]. Embedded devices face a wide variety of attacks similar to always-connected server-class systems. Hence, their security must become a first-class concern.

We focus on a particularly vulnerable and constrained subclass of embedded systems—bare-metal systems. They execute a single statically linked binary image providing both the (operating) system functionality and application logic without privilege separation between the two. Bare-metal systems are not an exotic or rare platform: they are often found as part of larger systems. For example, smart phones delegate control over the lower protocol layers of WiFi and Bluetooth to a dedicated bare-metal System on a Chip (SoC). These components can be compromised to gain access to higher level systems, as demonstrated by Google P0's disclosure of vulnerabilities in Broadcom's WiFi SoC that enable gaining control of a smartphone's application processor [6]. This is an area of growing concern, as SoC firmware has proven to be exploitable [16, 15, 17].

Protecting bare-metal systems is challenging due to tight resource constraints and software design patterns used on these devices. Embedded devices have limited energy, memory, and computing resources and often limited hardware functionality to enforce security properties. For example, a Memory Management Unit (MMU) which is required for Address Space Layout Randomization [42] is often missing. Due to the tight constraints, the dominant programming model shuns abstractions, allowing *all* code to access *all* data and peripherals without any active mitigations. For example, Broadcom's WiFi SoC did *not* enable Data Execution Prevention. Even if enabled, the entire address space is readable/writable by the executing program, thus a single bug can be used to trivially disable DEP by overwriting a flag in memory.

[1] ACES is available as open-source at `https://github.com/embedded-sec/ACES`.

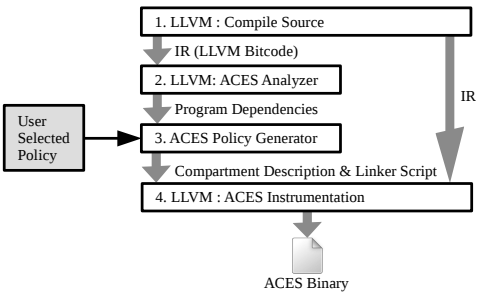

Figure 1: ACES's development tool flow overview.

Conventional security principles, namely, least privileges [45] or process isolation are challenging to implement in bare-metal systems. Bare-metal systems no longer focus on a dedicated task but increasingly run multiple independent or loosely related tasks. For example, a single SoC often implements both Bluetooth and WiFi, where neither Bluetooth nor WiFi needs to access the code and data of the other. However, without isolation, a single bug compromises the entire SoC and possibly the entire system [6].

While many bare-metal systems employ no defenses, there are ongoing efforts to improve their security. EPOXY [12] demonstrated how DEP, diversity, and stack protections can be deployed on bare-metal systems. However, EPOXY does not address the issue of least privileges or process isolation. MINION [27] uses the compiler and dynamic analysis to infer thread-level compartments and uses the OS's context switch to change compartments. It uses a fixed algorithm to determine the compartments, providing the developer no freedom in determining the best security boundaries for their application. ARM's Mbed μVisor [39] is a compartmentalization platform for ARM Cortex-M series devices. μVisor enables the developer to create compartments within a bare-metal application, thereby restricting access to data and peripherals to subsets of the code. It requires the developer to manually partition data and manage all communication between compartments. Compartments are restricted to individual threads, and all code is always executable, since no compartmentalization exists for code, only for data and peripherals. This results in a daunting challenge for developers, while only achieving coarse-grained data/peripheral compartments.

We present ACES (**A**utomatic **C**ompartments for **E**mbedded **S**ystems), an extension to the LLVM compiler that enables the exploration of strategies to apply the principle of least privileges to bare-metal systems. ACES uses two broad inputs—a high level, generic compartmentalization policy and the program source code. Using these, it automatically applies the policy to the application while satisfying the program's dependencies (*i.e.,* ensuring code can access its required data) and the underlying hardware constraints. This enables the developer to focus on the high-level policy that best fits her goals for performance and security isolation. Likewise, the automated workflow of ACES frees the developer from challenging implementation issues of the security controls.

Our work breaks the coupling between the application, hardware constraints, and the security policy, and enables the automatic enforcement of compartmentalization policies. ACES allows the formation of compartments based on functionality, *i.e.,* distinct functionality is separated into different compartments. It uses a piece of hardware widely available in even the low-end embedded devices called the Memory Protection Unit (MPU) to enforce access protections to different segments of memory from different parts of code. ACES moves away from the constraint in MINION and μVisor that an entire process or thread needs to be at the same privilege level. ACES extends the LLVM tool-chain and takes the policy specification as user input, as shown in Figure 1. It then creates a Program Dependence Graph (PDG) [21] and transforms compartmentalization into a graph partitioning problem. The result of the compilation pipeline is a secure binary that runs on the bare-metal device. We evaluate three policies to partition five IoT applications. The results demonstrate the ability to partition applications into many compartments (ranging from 14 to 34) protecting the integrity of data and restricting code reuse attacks. The policies have modest runtime overhead, on average 15.7% for the strongest policy.

In summary, our contributions are: (1) Integrity of code and data for unmodified applications running on bare-metal embedded devices. (2) Automated enforcement of security compartments, while maintaining program dependencies and respecting hardware constraints. The created compartments separate code and data, on a sub-thread level, breaking up the monolithic memory space of bare-metal applications. (3) Use of a micro-emulator to allow selective writes to small data regions. This eases restrictions on compartmentalization imposed by the MPU's limited number of regions and their size. (4) Separating the compartmentalization policy from the program implementation. This enables exploration of security-performance trade-offs for different compartmentalization policies, without having to rewrite application code and handle low level hardware requirements to enforce the policy.

2 Threat Model and Assumptions

We assume an attacker who tries to gain arbitrary code execution with access to an arbitrary read/write primitive. Using the arbitrary read/write primitive, the attacker can perform arbitrary malicious execution, *e.g.,* code in-

jection (in executable memory) or code reuse techniques (by redirecting indirect control-flow transfers [47]), or directly overwrite sensitive data. We assume that the software itself is trustworthy (*i.e.,* the program is buggy but not malicious). Data confidentiality defenses [11] are complementary to our work. This attacker model is in line with other control-flow hijack defenses or compartmentalization mechanisms.

We assume the system is running a single statically linked bare-metal application with no protections. We also assume the hardware has a Memory Protection Unit (MPU) and the availability of all source code that is to be compartmentalized. Bare-metal systems execute a single application, there are no dynamically linked or shared libraries. Lack of source code will cause a reduction in precision for the compartmentalization for ACES.

ACES applies defenses to: (1) isolate memory corruption vulnerabilities from affecting the entire system; (2) protect the integrity of sensitive data and peripherals. The compartmentalization of data, peripherals, *and* code confines the effect of a memory corruption vulnerability to an isolated compartment, prohibiting escalation to control over the entire system. Our threat model assumes a powerful adversary and provides a realistic scenario of current attacks.

3 Background

To understand ACES' design it is essential to understand some basics about bare-metal systems and the hardware on which they execute. We focus on the ARMv7-M architecture [3], which covers the widely used Cortex-M(3, 4, and 7) micro-controllers. Other architectures are comparable or have more relaxed requirements on protected memory regions simplifying their use [2, 5].

Address Space: Creating compartments restricts access to code, data, and peripherals during execution. Figure 2 shows ARM's memory model for the ARMv7-M architecture. It breaks a 32bit (4GB) memory space into several different areas. It is a memory mapped architecture, meaning that all IO (peripherals and external devices) are directly mapped into the memory space and addressed by dereferencing memory locations. The architecture reserves large amounts of space for each area, but only a small portion of each area is actually used. For example, the Cortex-M4 (STM32F479I) [48] device we use in our evaluation has 2MB of Flash in the code area, 384KB of RAM, and uses only a small portion of the peripheral space—and this is a higher end Cortex-M4 micro-controller. The sparse layout requires each area to have its own protection scheme.

Memory Protection Unit: A central component of compartment creation is controlling access to memory. ACES utilizes the MPU for this purpose. The MPU enables setting permissions on regions of physical memory. It controls read, write, and execute permissions for both privileged and unprivileged software. An MPU is similar to an MMU, however it does not provide virtual memory address translation. On the ARMv7-M architecture the MPU can define up to eight regions, numbered 0-7. Each region is defined by setting a starting address, size, and permissions. Each region must be a power of two in size, greater than or equal to 32 bytes and start at a multiple of its size (*e.g.,* if the size is 1KB then valid starting address are 0, 1K, 2K, 3K, etc). Each region greater than 256 bytes can be divided into eight equally sized sub-regions that are individually activated. All sub-regions have the same permissions. Regions can overlap, and higher numbered regions have precedence. In addition to the regions 0-7, a default region with priority -1 can be enabled for privileged mode only. The default region enables read, write, and execute permissions to most of memory. Throughout this paper, we use the term, "MPU region" to describe a contiguous area of memory whose permissions are controlled by one MPU register.

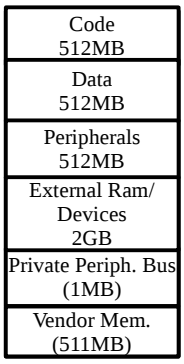

Figure 2: ARM's memory model for ARMv7-M devices

The MPU's restrictions significantly complicate the design of compartments. The limited number of regions requires all code, global variables, stack data, heap data, and peripherals that need to be accessed within a compartment to fit in eight contiguous regions of memory. These regions must satisfy the size and alignment restrictions of the MPU. The requirement that MPU region sizes be a power of two leads to fragmentation, and the requirement that MPU regions be aligned on a multiple of its size creates a circular dependency between the location of the region and its size.

Execution Modes: ARMv7-M devices support privileged and unprivileged execution modes. Typically, when executing in privileged mode, all instructions can be executed and all memory regions accessed. Peripherals, which reside on the private peripheral bus, are only accessible in privileged mode. Exception handlers always execute in privileged mode, and unprivileged code can create a software exception by executing an SVC

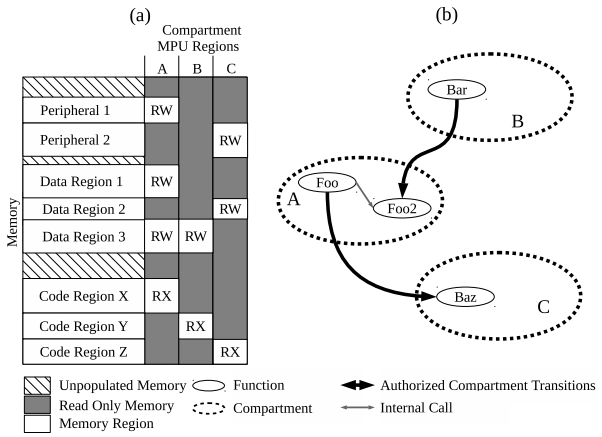

Figure 3: Illustration of ACES' concept of compartments. ACES isolates memory (a) – with permissions shown in the column set – and restricts control-flow between compartments (b).

(*i.e.,* supervisor call) instruction. This will cause the SVC exception handler to execute. This is the mechanism through which system calls are traditionally created in an OS. Bare-metal systems traditionally execute all code in privileged mode.

4 Design

ACES automatically enforces the principle of least privileges on bare-metal applications by providing write and control-flow integrity between regions of the program, *i.e.,* if a given code region is exploited via a vulnerability in it, the attack is contained to that compartment. A secondary goal of ACES is enabling a developer to explore compartmentalization strategies to find the correct tradeoffs between performance and security, without needing her to change the application.

4.1 PDG and Initial Region Graph

A compartment is defined as an isolated code region, along with its accessible data, peripherals, and allowed control-flow transfers. Each instruction belongs to exactly one compartment, while data and peripherals may be accessible from multiple compartments. Thus, our compartments are code centric, not thread centric, enabling ACES to form compartments within a single thread. Figure 3 shows several compartments, in it Compartment A enables access to code region X and read-write access to peripheral 1, data region 1, and data region 3. Compartment A can also transition from `Foo` into compartment C by calling `Baz`. Any other calls outside of the compartment are prohibited. When mapped to memory, a compartment becomes a region of contiguous code, and zero or more regions of data and peripherals. ACES utilizes the MPU to set permissions on each region and thus, the compartments must satisfy the MPU's constraints, such as starting address and number of MPU regions.

The starting point to our workflow is a Program Dependence Graph (PDG) [21]. The PDG captures all control-flow, global data, and peripheral dependencies of the application. While precise PDGs are known to be infeasible to create—due to the intractable aliasing problem [43], over approximations can be created using known alias analysis techniques (*e.g.,* type-based alias analysis [33]). Dynamic analysis gives only true dependencies and is thus more accurate, with the trade off that it needs to be determined during execution. ACES' design allows PDG creation using static analysis, dynamic analysis, or a hybrid.

Using the PDG and a device description, an initial *region graph* is created. The region graph is a directed graph that captures the grouping of functions, global data, and peripherals into MPU regions. An initial region graph is shown in Figure 4b, and was created from the PDG shown in Figure 4a. Each vertex has a type that is determined by the program elements (code, data, peripheral) it contains. Each code vertex may have directed edges to zero or more data and/or peripheral vertices. Edges indicate that a function within the code vertex reads or writes a component in the connected data/peripheral vertices.

The initial region graph is created by mapping all functions and data nodes in the PDG along with their associated edges directly to the region graph. Mapping peripherals is done by creating a vertex in the region graph for each edge in the PDG. Thus, a unique peripheral vertex is created for every peripheral dependency in the PDG. This enables each code region to independently determine the MPU regions it will use to access its required peripherals. The initial region graph does not consider hardware constraints and thus, applies no bounds on the total number of regions created.

4.2 Process for Merging Regions

The initial region graph will likely not satisfy performance and resource constraints. For example, it may require more data regions than there are available MPU regions, or the performance overhead caused by transitioning between compartments may be too high. Several regions therefore have to be merged. Merging vertices causes their contents to be placed in the same MPU region. Only vertices of the same type may be merged.

Code vertices are merged by taking the union of their contained functions and associated edges. Merging code vertices may expose the data/peripheral to merged func-

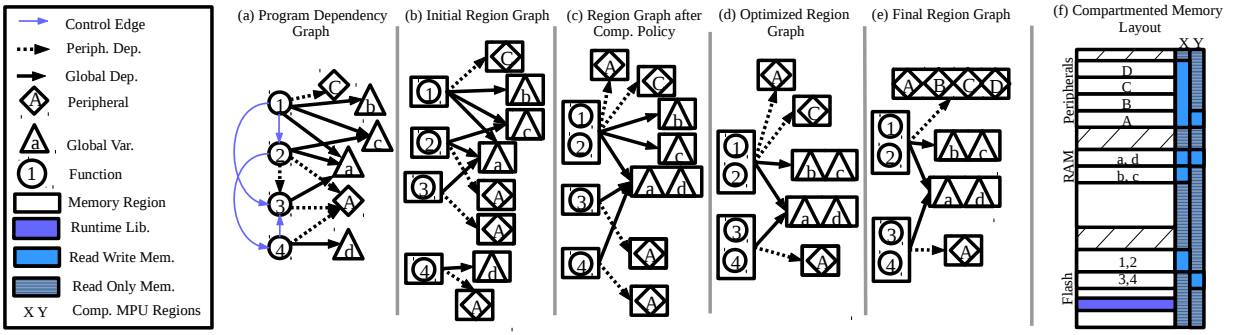

Figure 4: Compartment creation process and the resulting memory layout. (a) PDG is transformed to an initial region graph (b). A compartmentalization policy is applied (c), followed by optimizations (d) and lowering to produce the final region graph (e). Which, is mapped to a compartmented memory layout with associated MPU regions (f).

tions, as the compartment encompasses the union of all its contained function's data/peripheral dependencies. However, it improves performance as more functions are located in the same compartment. Similar to merging code vertices, merging of data vertices takes the union of the contained global variables and the union of their edges. All global variables in a vertex are made available to all dependent code regions. Thus, merging two data vertices increases the amount of code which can access the merged data vertices.

Unlike code and global variables, which can be placed anywhere in memory by the compiler, peripheral addresses are fixed in hardware. Thus, ACES uses a device description to identify all peripherals accessible when the smallest MPU region that covers the two merged peripherals is used. The device description contains the address and size of each peripheral in the device. Using the device description peripheral vertices in the PDG can be mapped to a MPU region which gives access to the peripheral. To illustrate, consider two peripherals vertices that are to be merged and a device description containing four peripherals A, B, C, and D at addresses 0x000, 0x100, 0x200, and 0x300 all with size 0x100. The first vertex to be merged contains peripheral B at address 0x100 and the second Peripheral D at address 0x300. The smallest MPU region that meets the hardware restrictions (*i.e.,* is a power of 2 aligned on a multiple of its size) covers addresses 0x000-0x3FF, and thus enables access to peripherals A-D. Thus, the vertex resulting from merging peripherals B and D, will contain peripherals A, B, C, and D.

4.3 Compartmentalization Policy and Optimizations

The compartment policy defines how code, global variables, and peripherals should be grouped into compartments. An example of a security-aware policy is grouping by peripheral, *i.e.,* functions and global variables are grouped together based on their access to peripherals. ACES does not impose restriction on policy choice. Obviously, the policy affects the performance and isolation of compartments, and, consequently, the security of the executable binary image. For example, if two functions which frequently call each other are placed in different code compartments then compartment transitions will occur frequently, increasing the overhead. From a security perspective, if two sets of global variables $\vec{V_1}$ and $\vec{V_2}$ are placed in the same compartment and in the original program code region C_1 accessed $\vec{V_1}$ and C_2 accessed $\vec{V_2}$ then unnecessary access is granted—now both code regions can access the entire set of variables. ACES enables the developer to explore the performance-security trade-offs of various policies.

After applying the compartmentalization policy, it may be desirable to adjust the resulting compartments. These adjustments may improve the security or the performance of the resulting compartmented binary. For example, if performance is too slow it may be desirable to merge regions to reduce compartment transitions. To accommodate this, we enable adjustment passes to be applied to the region graph after the compartment formation. Developer-selected optimizations may be applied to the region graph. An example of an optimization is the transformation from Figure 4c to Figure 4d. It merges functions 3 and 4 because they access the same memory regions and peripherals. After the optimizations are applied, the resulting region graph is lowered to meet hardware constraints.

4.4 Lowering to the Final Region Graph

Lowering is the process by which ACES ensures all formed compartments meet the constraints of the targeted hardware. As each compartment consists of a single code vertex and its peripherals and data vertex. Each

code vertex's out degree must be lower or equal to the number of available MPU regions because the number of access permissions that can be enforced is upper bounded by that. Any code region not meeting this criteria is lowered, by merging its descendant data and peripheral vertices until its out-degree is less than or equal to the cap. ACES does this iteratively, by merging a pair of data or peripheral vertices on each iteration. The vertices to merge are determined by a cost function, with the lowest cost merge being taken. Examples of cost functions include: the number of functions that will gain access to unneeded variables in the data regions, how security critical a component is (resulting in a high cost of merging), and the cost of unneeded peripherals included in the merge of two peripherals.

4.5 Program Instrumentation and Compartment Switching

ACES sets up the MPU configuration to isolate address spaces of individual processes, similar to how a desktop operating system handles the MMU configuration. ACES generates the appropriate MPU configuration from the final region graph and inserts code during a compilation pass to perform compartment transitions. Ensuring that the proper MPU configuration is used for each compartment is done by encoding each compartment's MPU configuration into the program as read-only data and then on each compartment transition, the appropriate configuration is loaded into the MPU.

Inserting compartment transitions requires instrumenting every function call between compartments and the associated return to invoke a compartment switching routine. Each call from one compartment into another has associated metadata listing the valid targets of the transition. For indirect function calls, the metadata lists all possible destinations. At runtime, the compartment switching routine decides if the transition is valid using this metadata. If authorized, it saves the current MPU configuration and return address to a *"compartment stack"*, and then configures the MPU for the new compartment. It then completes the call into the new compartment. On the associated return, the compartment stack is used to authenticate the return and restore the proper MPU configuration. The MPU configuration, compartment stack, and compartment switching routine are only writable by privileged code.

4.6 Micro-emulator for Stack Protection

The final element of ACES is stack protection. The constraints of MPU protection (starting address, size) mean that it is difficult to precisely protect small data regions and regions that cannot be easily relocated, such as the stack. To overcome these limitations we use a micro-emulator. It emulates writes to locations prohibited by the MPU regions, by catching the fault cause by the blocked access. It then emulates, in software, all the effects of the write instruction, *i.e.,* updates memory, registers, and processor flags. A white-list is used to restrict the areas each compartment is allowed to write.

An MPU region is used to prevent writing all data above the stack pointer on the stack. Thus, the entered compartment is free to add to the stack and modify any data it places on the stack. However, any writes to previous portions of the stack will cause a memory access fault. Then the micro-emulator, using a white-list of allowed locations, enables selective writes to data above the stack pointer.

To generate the white-list, static or dynamic analysis may be used. With static analysis large over approximations to available data would be generated. Whereas dynamic analysis may miss dependencies, potentially leading to broken applications. To support dynamic analysis, the emulator supports two modes of operation: record and enforce. In record mode, which happens in a benign training environment, representative tests are run and all blocked writes emulated and recorded on a per compartment basis. The recorded accesses create a white-list for each compartment. When executing in enforce mode (*i.e.,* after deployment) the micro-emulator checks if a blocked access is allowed using the white-list and either emulates it or logs a security violation. Significant use of dynamically allocated data on desktop systems would make dynamic analysis problematic. However, the limited memory on bare-metal systems requires developers to statically allocate memory, enabling dynamic analysis to readily identify data dependencies.

5 Implementation

ACES is implemented to perform four steps: program analysis, compartment generation, program instrumentation, and enforcement of protections at runtime. Program analysis and program instrumentation are implemented as new passes in LLVM 4.0 [32] and modifications to its ARM backend. Compartment generation is implemented in Python leveraging the NetworkX graph library [25]. Runtime enforcement is provided in the form of a C runtime library. For convenience, we wrap all these components with a Makefile that automates the entire process.

5.1 Program Analysis

Our program analysis phase creates the PDG used to generate the region graph, and is implemented as an IR pass in LLVM. To create the PDG it must identify control flow, global variable usage, and peripheral dependencies

for each function. Control-flow dependencies are identified by examining each call instruction and determining its possible destinations using type-based alias analysis [33]. That is, we assume an indirect call may call any function matching the function type of the call instruction. This identifies all potential control-flow dependencies, but generates an over-approximation.

Over-approximations of global variable accesses result in overly permissive compartments. We found that LLVM's alias analysis techniques give large over-approximations to data dependencies. Thus, we generate an under-approximation of the global variables that are accessed within each function using LLVM's use-def chains. We form compartments with this under-approximation and then use the micro-emulator to authenticate access to missed dependencies at runtime (Section 4.6). To understand our peripheral analysis, recall that the ARMv7-M architecture is a memory mapped architecture. This means regular loads and stores to constant addresses are used to access peripherals. In software this is a cast of a constant integer to a pointer, which is then dereferenced. ACES uses the cast and dereference as a heuristic to identify dependencies on peripherals. Using these analyses, ACES creates a PDG suitable for compartmentalization.

5.2 Compartment Creation

Compartment creation uses the PDG, a compartmentalization policy, and the target device description to create a final region graph. It is implemented in Python using the NetworkX [25] graph library, which provides the needed graph operations for ACES (like traversal and merging). By separating this component from LLVM, we enable the rapid investigation of different compartmentalization policies without having to manage the complexities of LLVM. Policies are currently implemented as a python function. Creating a new policy requires writing a graph traversal algorithm that merges regions based on desired criteria. We envision that the research community could develop these policies, and an application developer would select a policy much like they select compiler optimizations today.

The region graph is created from the PDG as outlined in Section 4.1. However, the nuances of handling peripherals justify further explanation. Peripherals are merged using the device description to build a tree of all the possible valid MPU regions that cover the device peripherals, called the *"device tree"*. In the device tree, the peripherals are the leaves and the interior nodes are MPU regions that cover all their descendant peripherals. For example, if peripheral $P1$ is at memory-mapped address $[\alpha, \alpha + \Delta_1]$ and peripheral $P2$ is at address $[\beta, \beta + \Delta_2]$, then the intermediate node immediately above it will allow access to addresses $[\alpha, \beta + \Delta_2]$. Thus, the closer to the root a node is, the larger the MPU region and the more peripherals it covers. Using this tree, the smallest possible merge between two peripherals can be found by finding their closest common ancestor. The device tree also identifies peripherals on the private peripheral bus which requires access from privileged mode. Code regions dependent on these peripherals must execute in privileged mode; for security, the number and size of such regions should be limited by the policy.

To start, we implement two compartmentalization policies, *"Peripheral"* and *"Filename"*. The `Peripheral` policy is a security policy that isolates peripherals from each other. Thus for an attack to start by exploiting one peripheral and affect another (*e.g.,* compromising a WiFi SOC to get to the application processor) multiple compartments would have to be traversed. The policy initially gives each code vertex adjacent to one or more peripherals in the PDG a unique color. Two code vertices adjacent to the same set of peripherals get the same color. It then proceeds in rounds, and in each round any code vertex with a control-flow dependency on vertices of only one color is given the same color. Rounds continue until no code vertices are merged, at which point all uncolored code vertices are merged into a single vertex. The `Filename` policy is a naïve policy that demonstrates the versatility of the policies ACES can apply, and pitfalls of such a policy. It groups all functions and global variables that are defined in the same file into the same compartment.

Two optimizations to the region graph can be applied after applying the `Filename` policy. Merging all code regions with identical data and peripheral dependencies, this reduces compartment transitions at runtime without changing data accessible to any compartments. The second optimization examines each function and moves it to the region that it has the most connections to, using the PDG to count connections. This improves the performance of the application by reducing the number of compartment transitions. By applying these two optimizations to the `Filename` policy we create a third compartmentalization policy, *"Optimized Filename"*.

After applying optimizations, the region graph is lowered to meet hardware constraints. For our experimental platform, this ensures that no code vertex has more than four neighboring data/peripheral vertices. While the MPU on our target ARMv7-M devices has eight regions, two regions are used for global settings, *i.e.,* making all memory read-only and enabling execution of the default code region, as will be explained in Section 5.3. Stack protection and allowing execution of the code vertex in the current compartment each requires one MPU region. This leaves four MPU regions for ACES to use to enable access to data and peripheral regions. Every code vertex

with an out-degree greater than four iteratively merges data or peripheral vertices until its out-degree is less than or equal to four. After lowering, the final region graph is exported as a JSON file, which the program instrumentation uses to create the compartments.

5.3 Program Instrumentation

Program instrumentation creates a compartmentalized binary, using the final region graph and the LLVM bitcode generated during program analysis. It is implemented by the addition of a custom pass to LLVM and modifications to LLVM's ARM backend. To instrument the program, all compartment transitions must be identified, each memory region must be placed so the MPU can enforce permissions on it, and the MPU configuration for each region must be added.

Using the final region graph, any control edge with a source and destination in different compartments is identified as a compartment transition. We refer to the function calls that cause a transition as *compartment entries*, and their corresponding returns as *compartment exits*. Each compartment transition is instrumented by modification to LLVM's ARM backend. It associates metadata to each compartment entry and replaces the call instruction (*i.e.,* BL or BLX on ARM) with an SVC instruction. The return instructions of any function that *may* be called by a compartment entry are replaced with an SVC instruction. The SVC instruction invokes the compartment switching routine, which changes compartments and then, depending on the type of SVC executed, completes the call or return.

The compartment pseudo code for the compartment switching routine is shown in Algorithm 1, and is called by the SVC handler. It switches compartments by reconfiguring the MPU, and uses a compartment stack to keep track of the compartment entries and exits. This stack is never writable by the compartment, protecting it from unauthorized writes. The stack also enables determining if a compartment entry needs to change compartments or just return to the existing compartment. This is needed because functions with an instrumented return can be called from within and outside of a compartment. When called from within a compartment there will be no entry on the compartment stack. Thus, if the return address does not match the top of the compartment stack, the compartment switching routine exits without modifying the MPU configuration. This also results in the compartment exit routine executing more frequently than the compartment entry routine, as seen in Figure 5.

While, LLVM can instrument source code it compiles, it cannot instrument pre-compiled libraries. Ideally, all source code would be available, but as a fallback, ACES places all pre-compiled libraries and any functions they call in an always executable code region. When called, this code executes in the context of the callee. Thus, the data writable by the library code is restricted to that of the calling compartment. This is advantageous from a security perspective, as it constrains the libraries' access to data/peripherals based on the calling context. We envision in the future libraries could be distributed as LLVM bitcode instead of machine code, enabling ACES to analyze and instrument the code to create compartments.

After instrumenting the binary, ACES lays out the program in memory to enable the MPU to enforce permissions. The constraints of the MPU in our target platform require that each MPU region be a power of two in size and the starting address must be a multiple of its size. This introduces a circular dependency between determining the size of a region and its layout in memory. ACES breaks this dependency by using two linker scripts sequentially. The first ignores the MPU restrictions and lays out the regions contiguously. The resulting binary is used to determine the size of all the regions. After the sizes are known, the second linker script expands each region to a power of two and lays out the regions from largest to smallest, starting at the highest address in Flash/RAM and working down. This arrangement minimizes the memory lost to fragmentation, while enabling each region to be located at a multiple of its size. ACES then generates the correct MPU configuration for each region and uses the second linker script, to re-compile the program. The MPU configuration is embedded into read-only memory (Flash), protecting it against attacks that modify the stored configuration in an attempt to change access controls. The output of the second linker script is a compartmented binary, ready for execution.

5.4 Micro-emulator for Stack Protection

The micro-emulator enables protection of writes on the stack, as described earlier in Section 4.6. The MPU restrictions prohibits perfect alignment of the MPU region to the allocated stack when entering a compartment. Thus, some portions of the allocated stack may remain accessible in the entered compartment. To minimize this, we disable as many sub-regions of the MPU as possible, while still allowing the current compartment to write to all the unallocated portions of the stack. With less restrictive MPUs—*e.g.,* the ARMv8-M MPU only requires regions be multiples of 32 bytes in size and aligned on a 32 byte boundary—this stack protection becomes stronger. In addition, the micro-emulator handles all writes where our static analysis under approximates and enables access to areas smaller than the MPU's minimum region size.

The micro-emulator can be implemented by modifying the memory permissions to allow access to the fault-

Algorithm 1 Compartment Switching Procedure

```
1: procedure CHANGE COMPARTMENTS
2:     Lookup SVC Number from PC
3:     if SVC 100 then                                  ▷ Compartment Entry
4:         Look up Metadata from PC
5:         if Target in Metadata then                   ▷ Target Addr. in LR
6:             Get MPU Config from Metadata for Target
7:         else
8:             Fault
9:         end if
10:        Set MPU Configuration
11:        Fixup Ret. Addr. to Skip Over Metadata
12:        Push Stack MPU Config to Comp. Stack
13:        Push Fixed Up Ret. Addr. to Comp. Stack
14:        Adjust Stack MPU region
15:        Fixup Stack to Exit into Target
16:        Exit SVC
17:    else if SVC 101 then                             ▷ Compartment Entry
18:        if Ret. Addr is on Top of Comp. Stack then
19:            Get Return MPU Config using LR
20:            Set MPU Config
21:            Pop Comp. Stack
22:            Pop Stack MPU Config
23:            Restore previous Stack MPU Config
24:        end if
25:        Fixup Stack to Exit to Ret. Addr.
26:        Exit SVC
27:    else
28:        Call Original SVC
29:    end if
30: end procedure
```

ing location and re-executing the store instruction, or emulating the store instruction in software. Re-executing requires a way to restore the correct permissions *immediately after* the store instruction executes. Conceptually, instruction rewriting, copying the instruction to RAM, or using the debugger to set a breakpoint can all achieve this. However, code is in Flash preventing rewriting instructions; copying the instruction to RAM requires making RAM writable and executable, thus exposing the system to code injection attacks. This leaves the debugger. However, on ARMv7-M devices, it can only be used by the internal software *or* an external debugger, not both. Using the debugger for our purpose prevents a developer from debugging the application. Therefore, we choose to emulate the write instructions in software.

The micro-emulator is called by the MemManage Fault handler, and emulates all the instructions that write to memory on the ARMv7-M architecture. As the emulator executes within an exception, it can access all memory. The handler emulates the instruction by performing all the effects of the instruction (*i.e.,* writing to memory and updating registers) in its original context. When the handler exits, the program continues executing as if the faulting instruction executed correctly. The emulator can be compiled in *record* or *enforce* mode. In record mode (used during training for benign runs), the addresses of all emulated writes are recorded on a per compartment basis. This allows the generation of the white-list for the allowable accesses. The white-list contains start and stop address for every addresses accessible through the emulator for each compartment. When generating the list, any recorded access to a global variable is expanded to allow access to all addresses. For example, if a single address of a buffer is accessed, the white list will contain the start and stop address for the entire buffer. The current emulator policy therefore grants access at variable granularity. This means the largest possible size of all variables does not have to be exercised during the recording runs. However, as peripherals often have memory mapped configuration register (*e.g.,* setting clock sources) and other registers for performing is function (*e.g.,* sending data). The white-list only allows access to peripheral addresses that were explicitly accessed during recording. Thus, a compartment could configure the peripheral, while another uses it.

6 Evaluation

To evaluate the effectiveness of ACES we compare the Naïve Filename, Optimized Filename, and Peripheral compartmentalization policies. Our goal is not to identify the best policy, but to enable a developer to compare and contrast the security and performance characteristics of the different policies. We start with a case study to illustrate how the different compartmentalization policies impact an attacker. We then provide a set of static metrics to compare policies, and finish by presenting the policy's runtime and memory overheads. We also compare the ACES' policies to Mbed μVisor, the current state-of-the-art in protecting bare-metal applications.

For each policy, five representative IoT applications are used. They demonstrate the use of different peripherals (LCD Display, Serial port, Ethernet, and SD card) and processing steps that are typically found in IoT systems (compute based on peripheral input, security functions, data sent through peripheral to communicate). **PinLock** represents a smart lock. It reads a pin number over a serial port, hashes it, compares it to a known hash, and if the comparison matches, sends a signal to an IO pin (akin to unlocking a digital lock). **FatFS-uSD** implements a FAT file system on an SD card. **TCP-Echo** implements a TCP echo server over Ethernet. **LCD-Display** reads a series of bitmaps from an SD card and displays them on the LCD. **Animate** displays multiple bitmaps from an SD card on the LCD, using multiple layers of the LCD to create the effect of animation. All except PinLock are provided with the development boards and written by STMicroelectronics. We create four binaries for each application, a baseline without any security enhancement, and one for each policy. PinLock executes on the STM32F4Discovery [49] development board and the others execute on the STM32F479I-Eval [48] development board.

6.1 PinLock Case Studies

To illustrate ACES' protections we use PinLock and examine ways an attacker could unlock the lock without entering the correct pin. There are three ways an attacker could open the lock using a memory corruption vulnerability. First, overwriting the global variable which stores the correct pin. Second, directly writing to the memory mapped GPIO controlling the lock. Third, bypassing the checking code with a control-flow hijack attack and executing the unlock functionality. We assume a write-what-where vulnerability in the function `HAL_UART_Receive_IT` that can be used to perform any of these attacks. This function receives characters from the UART and copies them into a buffer, and is defined in the vendor provided Hardware Abstraction Libraries (HAL).

Memory Corruption: We first examine how ACES impacts the attackers ability to overwrite the stored pin. For an attacker to overwrite the stored pin, the vulnerable function needs to be in a compartment that has access to the pin. This occurs when either the global variable is in one of the compartments' data regions or its whitelist. In our example, the target value is stored in the global variable `key`. In the Naïve Filename and Optimized Filename policies the only global variable accessible to `HAL_UART_Receive_IT`'s compartment is a UART Handle, and thus the attacker cannot overwrite `key`. With the peripheral policy `key` is in a data region accessible by `HAL_UART_Receive_IT`'s compartment. Thus, `key` can be directly overwritten. Directly writing the GPIO registers is similar to overwriting a global variable and requires write access to the GPIO-A peripheral. Which is not accessible to `HAL_UART_Receive_IT`'s compartment under any of the policies.

Control-Flow Hijacking: Finally, the attacker can unlock the lock by hijacking control-flow. We consider an attack to be successful if any part of the unlock call chain, shown in Listing 1, is executable in the same compartment as `HAL_UART_Receive_IT`. If this occurs, the attacker can redirect execution to unlock the lock illegally. We refer to this type of control-flow attack as direct, as the unlock call chain can be directly executed. For our policies, this is only possible with the Peripheral policy. This occurs because `HAL_UART_Receive_IT` and main are in the same compartment. For the other policies `HAL_UART_Receive_IT`'s compartment does not include any part of the unlock call chain. A second type of attack—a confused deputy attack—may be possible if there is a valid compartment switch in the vulnerable function's compartment to a point in the unlock call chain. This occurs if a function in the same compartment as the vulnerable function makes a call into the unlock call chain. This again only occurs with the Peripheral policy, as `main` contains a compartment switch into `unlock`'s compartment. A summary of the attacks and the policies protections against them is given in Table 1.

Listing 1: PinLock's unlock call chain and filename of each call

```
main // main.c
unlock // main.c
BSP_LED_On // stm32f401_discovery.c
HAL_GPIO_WritePin // stm32f4xx_hal_gpio.c
```

Table 1: Summary of ACES' protection on PinLock for memory corruption vulnerability in function HAL_UART_Receive_IT. (✓) – prevented, ✗– not prevented

Policy	Overwrite		Control Hijack	
	Global	GPIO	Direct	Deputy
Naïve Filename	✓	✓	✓	✓
Optimized Filename	✓	✓	✓	✓
Peripheral	✗	✓	✗	✗

6.2 Static Compartment Metrics

The effectiveness of the formed compartments depends on the applied policy. We examine several metrics of compartments that can be used to compare compartmentalization policies. Table 2 shows these metrics for the three compartmentalization policies. All of the metrics are calculated statically using the final region graph, PDG, and the binary produced by ACES.

Number of Instructions and Functions: The first set of metrics in Table 2 are the number of instructions and the number of functions used in the ACES binaries, with percent increase over baseline shown in parentheses. To recap, the added code implements: the compartment switching routine, instruction emulation, and program instrumentation to support compartment switching. They are part of the trusted code base of the program and thus represent an increased risk to the system that needs to be offset by the gains it makes in security. ACES' runtime support library is the same for all applications and accounts for 1,698 of the instructions added. The remaining instructions are added to initiate compartment switches. As many compartments are formed, we find in all cases the number of instructions accessible at any given point in execution is less than the baseline. This means that ACES is always reducing the code that is available to execute.

Reduction in Privileged Instructions: Compartmentalization enables a great reduction in the number of instructions that require execution in privileged mode, Table 2, shown as "% Priv.". This is because it enables

Table 2: Static Compartment Evaluation Metrics. Percent increase over baseline in parentheses for ACES columns.

Application	Policy	ACES			#Regions		Instr. Per Comp		Med. Degree		Exposure	#ROP
		#Instrs.	%Priv.	#Functs.	Code	Data	Med.	Max	In	Out	Med. #Stores.	Reduction
PinLock	Naïve Filename	8,374(50.9%)	2.9%	193(17.0%)	14	7	1,501	2,739	6	3	118 (11.0%)	345 (47.9%)
	Opt. Filename	8,332(50.1%)	26.2%	193(17.0%)	11	6	1,418	2,983	3	1	737 (68.8%)	341 (48.5%)
	Peripheral	8,342(50.3%)	9.8%	193(17.0%)	20	8	1,298	3,291	1	1	489 (45.7%)	345 (47.9%)
FatFs-uSD	Naïve Filename	21,222(18.4%)	1.2%	324(9.5%)	23	13	1,563	6,825	6	4	164 (4.2%)	432 (74.2%)
	Opt. Filename	21,083(17.6%)	15.0%	324(9.5%)	19	2	1,380	10,316	2	1	3,081 (79.6%)	709 (57.6%)
	Peripheral	21,096(17.7%)	3.4%	324(9.5%)	23	9	1,565	8,701	1	1	1,560 (40.4%)	699 (58.2%)
TCP-Echo	Naïve Filename	34,477(12.7%)	0.7%	445(6.7%)	37	23	1,789	5,058	26	8	256 (4.7%)	384 (85.3%)
	Opt. Filename	34,324(12.2%)	10.6%	445(6.7%)	28	4	1,476	14,395	23	3	3,970 (74.9%)	646 (75.2%)
	Peripheral	33,408(9.2%)	0.6%	445(6.7%)	23	11	1,198	23,100	1	1	3,327 (61.6%)	1,759 (32.5%)
LCD-uSD	Naïve Filename	38,806(12.1%)	0.6%	462(6.5%)	33	17	10,290	14,291	10	4	93 (1.5%)	1,173 (58.5%)
	Opt. Filename	38,452(11.1%)	19.7%	462(6.5%)	25	5	10,006	15,499	7	3	3,500 (59.5%)	1,385 (51.0%)
	Peripheral	38,109(10.1%)	1.9%	462(6.5%)	34	15	9,900	17,188	2	2	3,247 (55.0%)	1,524 (46.0%)
Animation	Naïve Filename	38,894(12.1%)	0.6%	466(6.4%)	33	16	10,265	14,246	10	5	105 (1.7%)	1,178 (58.7%)
	Opt. Filename	38,499(10.9%)	28.7%	466(6.4%)	23	3	9,954	18,317	6	3	4,257 (72.5%)	1,401 (50.8%)
	Peripheral	38,194(10.1%)	1.9%	466(6.4%)	34	17	9,850	19,015	2	2	2,498 (42.1%)	1,568 (45.0%)

only the code which accesses the private peripheral bus and the compartment transition logic to execute in privileged mode. The Naïve Filename and Peripheral policy show the greatest reductions, because of the way they form compartments. As only a small number of functions access the private peripheral bus—defined in a few files—the Naïve Filename creates small compartments with privileged code. The Optimized Filename starts from the Naïve policy and then merges groups together, increasing the amount of privileged code, as privileged code is merged with unprivileged code. Finally, the Peripheral policy identifies the functions using the private peripheral bus. It then merges the other functions that call or are called by these functions and that have no dependency on any other peripheral. The result is it a small amount of privileged code.

Number of Regions: Recall a compartment is a single code region and collection of accessible data and peripherals. The number of code and data regions created indicates how much compartmentalization the policy creates. As the number of compartments increases, additional control-flow validation occurs at runtime as compartment transitions increase. Generally, larger numbers of regions indicate better security.

Instructions Per Compartment: This metric measures how many instructions are executable at any given point in time, and thus usable in a code reuse attack. It is the number of instructions in the compartment's code region plus the number of instructions in the default code region. Table 2 shows the median, and maximum number of instructions in each compartment. For all policies, the reduction is at least 23.9% of the baseline application, which occurs on TCP-Echo with the Peripheral policy. The greatest (83.4%) occurs on TCP-Echo with the Naïve Filename policy, as the TCP stack and Ethernet driver span many files, resulting in many compartments. However, the TCP stack and Ethernet driver only use the Ethernet peripheral. Thus, the Peripheral policy creates a large compartment, containing most of the application.

Compartment Connectivity: Compartment connectivity indicates the number of unique calls into (In Degree) or out of a compartment (Out Degree), where a unique call is determined by its source and destination. High connectivity indicates poor isolation of compartments. Higher connectivity indicates increasing chances for a confused deputy control-flow hijack attack between compartments. The ideal case would be many compartments with minimal connectivity. In all cases, the Naïve Filename policy has the worst connectivity because the applications make extensive use of abstraction libraries, (*e.g.,* hardware, graphics, FatFs, and TCP). This results in many files being used with many calls going between functions in different files. This results in many compartments, but also many calls between them. The Optimized Filename policy uses the Naïve policy as a starting point and relocates functions to reduce external compartment connectivity, but can only improve it so much. The Peripheral policy creates many small compartments with very little connectivity and one compartment with high connectivity.

Global Variable Exposure: In addition to restricting control-flow in an application, ACES reduces the number of instructions that can access a global variable. We measure the number of store instructions that can access a global variable—indicating how well least privileges are enforced. Table 2 shows the median number of store instructions each global variable in our test applications is writable from, along with the percent of store instructions in the application that can access it. Smaller numbers are better. The Filename policy has the greatest reduction in variable exposure. The other policies create larger data and code regions, and thus have increased variable exposure. In addition, lowering to four memory regions causes multiple global variables to be merged into the same data region, increasing variable exposure. Having more MPU regions (the ARMv8-M architecture

supports up to 16) can significantly improve this metric. As an example, we compiled Animation using the Optimized Filename policy and 16 MPU regions (lowering to 12 regions). It then creates 28 data regions versus three with eight MPU regions.

ROP Gadgets: We also measure the maximum number of ROP gadgets available at any given time during execution, using the ROPgadget compiler [46]. ROP gadgets are snippets of instructions that an attacker chains together to perform control-hijack attacks while only using existing code [47]. As shown in Table 2, ACES reduces the number of ROP gadgets between 32.5% and 85.3% compared to the baseline; the reduction comes from reducing the number of instructions exposed at any point during execution.

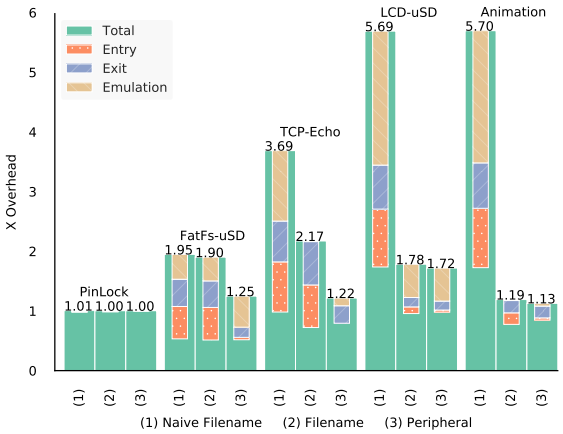

Figure 5: Runtime overhead for applications.

6.3 Runtime Overhead

Bare-metal systems have tight constraints on execution time and memory usage. To understand ACES' impact on these aspects across policies, we compare the IoT applications compiled with ACES against the baseline unprotected binaries. For applications compiled using ACES, there are three causes of overhead: compartment entry, compartment exit, and instruction emulation. Compartment entries and exits replace a single instruction with an SVC call, authentication of the source and the destination of the call, and then reconfiguration of the MPU registers. Emulating a store instruction replaces a single instruction with an exception handler, authentication, saving and restoring context, and emulation of the instruction.

In the results discussion, we use a linguistic shorthand—when we say "compartment exit" or simply "exit", we mean the number of invocations of the compartment exit routine. Not all such invocations will actually cause a compartment exit for the reason described in Section 5.3.

All applications—except TCP-Echo—were modified to start profiling just before main begins execution and stops at a hard coded point. Twenty runs of each application were averaged and in all cases the standard deviation was less than 1%. PinLock stops after receiving 100 successful unlocks, with a PC sending alternating good and bad codes as quickly as the serial communication allows. FatFS-uSD formats its SD card, creates a file, writes 1,024 bytes to the file, and verifies the file's contents, at which point profiling stops. LCD-uSD reads and displays 3 of the 6 images provided with the application, as quickly as possible. Profiling stops after displaying the last image. The Animation application displays 11 of the 22 animation images provided with the application before profiling stops. Only half the images were used to prevent the internal 32bit cycle counters from overflowing. For TCP-Echo, a PC directly connected to the board sends TCP packets as fast as possible. We start profiling after receiving 20 packets—to avoid measuring the PC's delay in identifying the IP address of the board—and measure receiving 1,000 packets. This enables an accurate profiling of ACES' overhead, omitting the initialization of the board, which performs many compartment transitions.

The performance results for the three policies are shown in Figure 5. It shows the total overhead, along with the breakdown of portion of time spent executing compartment entries, compartment exits, and emulating instructions. Perhaps unintuitive, the time spend executing these components does not always contribute to a proportional increase in total execution time. This is because the programs block on IO. ACES changes what it does while blocking, but not how long it blocks. This is particularly evident on PinLock which has no measurable increase in total execution time for any policy, yet executes over 12,000 compartment entries and exits with the Naïve and Optimized Filename policies. This is because the small percentage of the time it spends executing compartment switches is hidden by the time spent waiting to receive data on the relatively slow serial port. The other applications wait on the Ethernet, uSD card, or LCD. In some cases, the overhead is not all attributed to compartment entries, exits or emulated instructions, this is because our instrumentation causes a small amount of overhead (about 60 instructions) on each event. In the case of LCD-uSD with the Naïve policy which executes over 6.8 million compartment entries, exits, and emulator calls this causes significant overhead.

Looking across the policies and applications we see that the Naïve Filename policy has the largest impact on execution. This is because the programs are written using many levels of abstraction. Consider TCP-Echo: it is written as an application on top of the Lightweight IP

Library (LwIP) implementation of the TCP/IP stack [19] and the boards HAL. LwIP uses multiple files to implement each layer of the TCP stack and the HAL uses a separate file to abstract the Ethernet peripheral. Thus, while the application simply moves a received buffer to a transmit buffer, these function calls cause frequent compartment transitions, resulting in high overhead. The Optimized Filename policy improves the performance of all applications by reducing the number of compartment transitions and emulated instructions. This is expected as it optimizes the Naïve policy by moving functions to compartments in which it has high connectivity, thus reducing the number of compartment transitions. This also forms larger compartments, increasing the likelihood that needed data is also available in the compartment reducing the number of emulated calls. Finally, the Peripheral policy gives the best performance, as its control-flow aware compartmentalization creates long call chains within the same compartment. This reduces the number of compartment transitions. The stark difference in runtime increase between policies highlights the need to explore the interactions between policies and applications, which ACES enables.

6.4 Memory Overhead

In addition to runtime overhead, compartmentalization increases memory requirements by: including ACES's runtime library (compartment switcher, and micro-emulator), adding metadata, adding code to invoke compartment switches, and losing memory to fragmentation caused by the alignment requirements of the MPU. We measure the increase in flash, shown in Figure 6, and RAM, show in Figure 7, for the test applications compiled with ACES and compare to the baseline breaking out the overhead contributions of each component.

ACES increases the flash required for the runtime library by 4,216 bytes for all applications and policies.

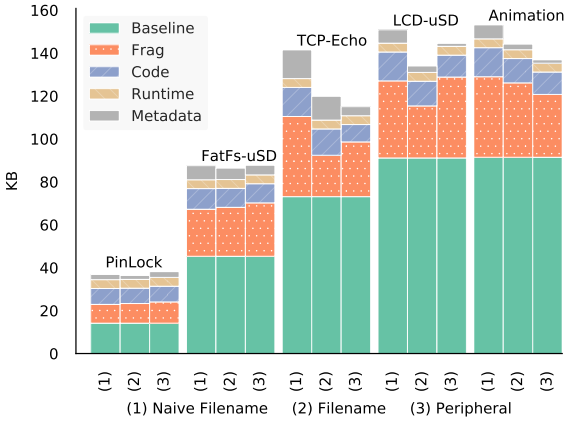

Figure 6: Flash usage of ACES for test applications

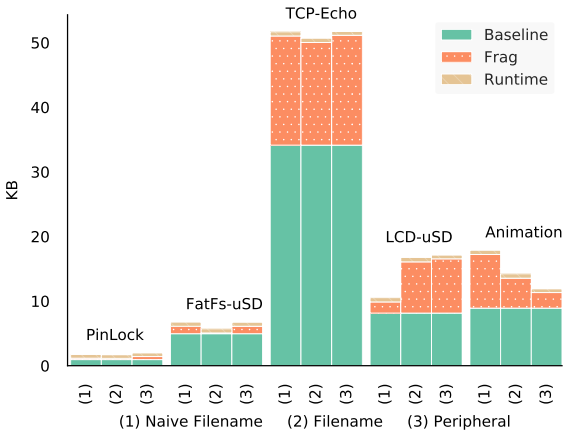

Figure 7: RAM usage of ACES for test applications

Fragmentation accounts for a significant amount of the increase in flash usage ranging from 26% of the baseline on Optimize Filename LCD-uSD to 70% on Peripheral PinLock. Fragmentation can also cause a large increase in RAM usage. This suggests that compartmentalization policies may need to optimize for fragmentation when creating compartments to reduce its impact. The MPU in the ARMv8-M architecture only requires regions be a multiple of 32 bytes and aligned on a 32 byte boundary. This will nearly eliminate memory lost to fragmentation on this architecture. For example, Peripheral TCP-Echo would only lose 490 bytes of flash and 104 bytes of RAM to padding versus 38,286 bytes and 17,300 bytes to fragmentation. Metadata and switching code increase are the next largest components, and are application and policy dependent. They increase with the number of compartment transitions and size of emulator white-lists.

Figure 7 shows the increase in RAM usage caused by ACES. Its only contributors to overhead are the runtime library and fragmentation. The runtime library consists of a few global variables (*e.g.,* compartment stack pointer), the compartment stack, and the emulator stack. The compartment stack—ranges from 96 bytes (Peripheral PinLock) to 224 bytes (Optimized Filename Animation)—and the emulator stack uses 400 bytes on all applications. Like flash, fragmentation can also cause a significant increase in RAM usage.

6.5 Comparison to Mbed μVisor

To understand how ACES compares to the state-of-the-art compartmentalization technique for bare-metal systems, we use the Mbed μVisor from ARM [39]. Mbed μVisor is a hypervisor designed to provide secure data and peripheral isolation between different compartments in the application (the equivalent term to compartment that μVisor uses is "box"). It is linked as a library to Mbed OS [38] and initialized at startup.

Table 3: Comparison of security properties between ACES and Mbed μVisor

Tool	Technique	DEP	Compartmentalization Type		
			Code	Data	Peripheral
ACES	Automatic	✓	✓	✓	✓
Mbed μVisor	Manual	✗(Heap)	✗	✓	✓

Table 3 shows a comparison of security protections provided by ACES and Mbed μVisor. Mbed μVisor requires manual annotation and specific μVisor APIs to be used to provide its protections, while ACES is automatic. Additionally, Mbed μVisor does not enforce code isolation, as all code is placed in one memory region that is accessible by *all* compartments. Furthermore, Mbed μVisor does not enforce DEP on the heap. Both enforce data and peripheral isolation among compartments. ACES enforces fine-grained compartmentalization by allowing code and data to be isolated within a thread, while Mbed μVisor requires a thread for each compartment with no isolation within a thread. Another advantage of ACES over Mbed μVisor is its compartments are not hard-coded into the application, enabling them to be automatically determined from high-level policies.

We compare ACES and Mbed μVisor by porting PinLock to Mbed μVisor. With μVisor, we used two compartments, which logically follows the structure of the application—one compartment handles the IO communication with the serial port and the other handles the computation, *i.e.,* the authentication of the pincode read from the serial port. The first has exclusive access to the serial port reading the user's pincode. The second compartment cannot access the serial port but can only request the entered pin from the first compartment. The authenticator then computes the hash and replies to the first compartment with the result. Mbed μVisor requires specific APIs and a main thread for each compartment, thus there is significant porting effort to get this (and any other application) to execute with μVisor. Table 4 shows a comparison between ACES and Mbed μVisor for Flash usage, RAM usage, runtime, and number of ROP gadgets. Since Mbed μVisor requires an OS, Flash and memory usage will be inherently larger. It allocates generous amounts of memory to the heap and stacks, which can be tuned to the application. For our comparison, we dynamically measure the size of the stacks and ignore heap size, thus under-reporting μVisor memory size. Averaged across all policies, ACES reduces the Flash usage by 58.6% and RAM usage by 83.9%, primarily because it does not require an OS.

ACES runtime is comparable (5.0% increase), thus ACES provides automated protection, increased compartmentalization, and reduced memory overhead with little impact on performance.

We investigate the security implications of having code compartmentalization by analyzing the number of ROP gadgets using the ROPgadget compiler [46]. Without code compartmentalization, a memory corruption vulnerability allows an attacker to leverage all ROP gadgets available in the application—the "Total" column in Table 4. Code compartmentalization confines an attacker to ROP gadgets available only in the current compartment. Averaged across all policies, ACES reduces the maximum number of ROP gadgets by 94.3% over μVisor.

Table 4: Comparison of memory usage, runtime, and the number of ROP gadgets between ACES and Mbed μVisor for the PinLock application.

Policy	Flash	RAM	Runtime # Cycles	# ROP Gadgets Total	Maximum	Average
ACES-Naïve Filename	33,504	4,712	526M	525	345 (53.2%)	234 (36.0%)
ACES-Opt. Filename	33,008	4,640	525M	671	341 (44.8%)	247 (32.4%)
ACES-Peripheral	34,856	5,136	525M	645	345 (47.2%)	204 (31.3%)
Mbed μVisor	81,604	30,004	501M	5,997	5,997 (100%)	5,997 (100%)

7 Related Work

Micro-kernels: Micro-kernels [35, 28] implement least privileges for kernels by reducing the kernel to the minimal set of functionality and then implement additional functions as user space "servers". Micro-kernels like L4 [35] have been successfully used in embedded systems [20]. They rely on careful development or formal verification [28] of the kernel and associated servers to maintain the principle of least privilege. ACES creates compartments within a single process, while micro-kernels break a monolithic kernel into many processes. In addition, the process of creating micro-kernels is manual while ACES' compartments are automatic.

Software Fault Isolation and Control-flow Integrity: Software fault isolation [50, 51] uses checks or pointer masking to restrict access of untrusted modules of a program to a specific region. SFI has been proposed for ARM devices using both software (ARMor) [55], and hardware features (ARMlock) [56]. ARMlock uses memory domains which are not available on Cortex-M devices. ACES works on micro-controllers and uses the MPU to ensure that code and data writes are constrained to a compartment without requiring pointer instrumentation. It also enables flexible definitions of what should be placed in each compartment whereas SFI assumes compartments are identified *a priori*.

Code Pointer Integrity [31] prevents memory corruptions from performing control flow hijacks by ensuring the integrity of code pointers. Control-flow integrity [1, 34, 53, 54, 41, 10] restricts the targets of indirect jumps to a set of authorized targets. This restricts the ability of an attacker to perform arbitrary execution, however arbitrary execution is still possible if a suffi-

ciently large number of targets are available to an attacker. ACES enforces control-flow integrity on control edges that transition between compartments. It also restricts the code and data available in each compartment, thus limiting the exposed targets at any given time.

Kernel and Application Compartmentalization: There has been significant work to isolate components of monolithic kernels using an MMU [57, 52, 18]. ACES focuses on separating a single bare-metal system into compartments using an MPU and addresses the specific issues that arise from the MPU limitations. Privtrans [9] uses static analysis to partition an application into privileged and unprivileged processes, using the OS to enforce the separation of the processes. Glamdring [36] uses annotations and data and control-flow analysis to partition an application into sensitive and non-sensitive partitions—executing the sensitive partition in an Intel SGX [13] enclave. Robinov *et al.* [44] partition Android applications into compartments to protect data and utilize ARM's TrustZone environment to run sensitive compartments. These techniques rely on an OS [9, 36] for process isolation or hardware not present on microcontrollers [36, 37, 44] or significant developer annotation [24, 36, 37]. In contrast ACES works without an OS, only requires an MPU, and does not require developer annotations.

Embedded system specific protections: NesCheck [40] provides isolation by enforcing memory safety. MINION [27] provides automatic thread-level compartmentalization, requiring an OS, while ACES provides function-level compartmentalization without an OS. ARM's TrustZone [4] enables execution of software in a "secure world" underneath the OS. TrustZone extensions are included in the new ARMv8-M architecture. At the time of writing, ARMv8-M devices are not yet available. FreeRTOS-MPU [22] is a real-time OS that uses the MPU to protect the OS from application tasks. Trustlite [29] proposes hardware extensions to microcontrollers, including an execution aware MPU, to enable the deployment of trusted modules. Each module's data is protected from the other parts of the program by use of their MPU. TyTan [7] builds on Trustlite and develops a secure architecture for low-end embedded systems, isolating tasks with secure IPC between them. In contrast, ACES enables intraprocess compartmentalization on existing hardware and separates compartment creation from program implementation.

8 Discussion and Conclusion

As shown in Section 6.3, compartmentalization policies may significantly impact runtime performance. To reduce the runtime impact, new policies should seek to place call chains together, and minimize emulating variable accesses. The PDG could be augmented with profiling information of baseline applications so that compartment policies can avoid placing callers and callees of frequently executed function calls in different compartments. In addition, the number of emulator calls could be reduced by improved alias analysis or adding dynamically discovered accesses to the PDG. This would enable an MPU region to be used to provide access to these variables. Finally, optimizations to the way emulated variables are accessed could be made to ACES. For example, the emulator could be modified to check if the store to be emulated is from memcpy. If so, permissions for the entire destination buffer could be validated and then the emulator could perform the entire buffer copy. Thus, the emulator would only be invoked once for the entire copy and not for each address written in the buffer.

Protecting against confused deputy attacks [26] is challenging for compartmentalization techniques. They use control over one compartment to provide unexpected inputs to another compartment causing it to perform insecure actions. Consider PinLock that is split into an unprivileged compartment and the privileged compartment with the unlock pin. An attacker with control over the unprivileged compartment may use any interaction between the two compartments to trigger an unlock event. To guard against confused deputy attacks, ACES restricts and validates the locations of all compartment transitions. The difficulty of performing these attacks depends on the compartmentalization policy. For security, it is desirable to have long compartment chains, resulting in many compartments that must be compromised to reach the privileged compartment.

In conclusion, ACES enables automatic application of compartments enforcing least privileges on bare-metal applications. Its primary contributions are (1) decoupling the compartmentalization policy from the program implementation, enabling exploration of the design space and changes to the policy after program development, *e.g.,* depending on the context the application is run in. (2) The automatic application of compartments while maintaining program dependencies and ensuring hardware constraints are satisfied. This frees the developer from the burden of configuring and maintaining memory permissions and understanding the hardware constraints, much like an OS does for applications on a desktop. (3) Use of a micro-emulator to authorize access to data outside a compartment's memory regions, allowing imprecise analysis techniques to form compartments. We demonstrated ACES's flexibility in compartment construction using three compartmentalization policies. Compared to Mbed μVisor, ACES' compartments use 58.6% less Flash, 83.9% less RAM, with comparable execution time, and reduces the number of ROP gadgets by an average of 94.3%.

9 Acknowledgments

We would like to thank Nathan Burow and Brian Hays for their careful reviews and input, and Brenden Dolan-Gavitt, our shepherd, for his detailed reviews and constructive input. This work was supported by Sandia National Laboratories, ONR award N00014-17-1-2513, NSF CNS-1513783, NSF CNS-1718637, NSF CNS-1548114, Intel Corporation, and Northrop Grumman Corporation through their Cybersecurity Research Consortium. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525. SAND2018-6917C

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *ACM Conf. on Computer and Communication Security* (2005), ACM, pp. 340–353.

[2] ARM. Armv8-m architecture reference manual. https://static.docs.arm.com/ddi0553/a/DDI0553A_e_armv8m_arm.pdf.

[3] ARM. Armv7-m architecture reference manual. https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf, 2014.

[4] ARM. Trustzone. http://www.arm.com/products/processors/technologies/trustzone/, 2015.

[5] ATMEL. Arm32 architecture document. https://www.mouser.com/ds/2/268/doc32000-1066014.pdf.

[6] BENIAMINI, G. Project Zero: Over The Air: Exploiting Broadcoms Wi-Fi Stack.

[7] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. Tytan: Tiny trust anchor for tiny devices. In *Design Automation Conf.* (2015), ACM/IEEE, pp. 1–6.

[8] BROCIOUS, C. My arduino can beat up your hotel room lock. In *Black Hat USA* (2013).

[9] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004), pp. 57–72.

[10] BUROW, N., CARR, S. A., NASH, J., LARSEN, P., FRANZ, M., BRUNTHALER, S., AND PAYER, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys 50*, 1 (2018, preprint: https://arxiv.org/abs/1602.04056).

[11] CARR, S. A., AND PAYER, M. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 193–204.

[12] CLEMENTS, A. A., ALMAKHDHUB, N. S., SAAB, K. S., SRIVASTAVA, P., KOO, J., BAGCHI, S., AND PAYER, M. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symp. on Security and Privacy* (2017), IEEE.

[13] COSTAN, V., AND DEVADAS, S. Intel sgx explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[14] CUNNINGHAM, M., AND MANNIX, L. Fines for red-light and speed cameras suspended across the state. *The Age* (06 2017).

[15] CVE-2017-6956. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6956, 2017.

[16] CVE-2017-6957. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6957, 2017.

[17] CVE-2017-6961. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6961, 2017.

[18] DAUTENHAHN, N., KASAMPALIS, T., DIETZ, W., CRISWELL, J., AND ADVE, V. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Conf. on Architectural Support for Programming Languages and Operating Systems* (2015), pp. 191–206.

[19] DUNKELS, A. Full tcp/ip for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (2003), ACM, pp. 85–98.

[20] ELPHINSTONE, K., AND HEISER, G. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 133–150.

[21] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9*, 3 (1987), 319–349.

[22] FreeRTOS-MPU. `http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html`.

[23] GREENBERG, A. The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse. *Wired Magazine* (08 2016).

[24] GUDKA, K., WATSON, R. N., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., MARINOS, I., NEUMANN, P. G., AND RICHARDSON, A. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1016–1031.

[25] HAGBERG, A. A., SCHULT, D. A., AND SWART, P. J. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)* (Pasadena, CA USA, Aug. 2008), pp. 11–15.

[26] HARDY, N. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review 22*, 4 (1988), 36–38.

[27] KIM, C. H., KIM, T., CHOI, H., GU, Z., LEE, B., ZHANG, X., AND XU, D. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp. (NDSS)* (2018).

[28] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[29] KOEBERL, P., SCHULZ, S., SCHULZ, P., SADEGHI, A., AND VARADHARAJAN, V. TrustLite: a security architecture for tiny embedded devices. *ACM EuroSys* (2014).

[30] KREBS, B. DDoS on Dyn Impacts Twitter, Spotify, Reddit. `https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/`.

[31] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code Pointer Integrity. *USENIX Symp. on Operating Systems Design and Implementation* (2014).

[32] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. Code Generation and Optimization* (2004), IEEE, pp. 75–86.

[33] LATTNER, C., LENHARTH, A., AND ADVE, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN 42*, 6 (2007), 278–289.

[34] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Trans. on Information Forensics and Security 6*, 4 (2011), 1404–1417.

[35] LIEDTKE, J. On micro-kernel construction. In *Symp. on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[36] LIND, J., PRIEBE, C., MUTHUKUMARAN, D., O'KEEFFE, D., AUBLIN, P.-L., KELBERT, F., REIHER, T., GOLTZSCHE, D., EYERS, D., KAPITZA, R., ET AL. Glamdring: Automatic application partitioning for intel sgx. In *USENIX Annual Technical Conf.* (2017).

[37] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1607–1619.

[38] mbed OS. `https://www.mbed.com/en/development/mbed-os/`.

[39] The mbed OS uVisor. `https://www.mbed.com/en/technologies/security/uvisor/`.

[40] MIDI, D., PAYER, M., AND BERTINO, E. Memory Safety for Embedded Devices with nesCheck. In *ACM Symp. on InformAtion, Computer and Communications Security* (2017).

[41] NIU, B., AND TAN, G. Modular control-flow integrity. *ACM SIGPLAN Notices 49*, 6 (2014), 577–587.

[42] PAX TEAM. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[43] RAMALINGAM, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994).

[44] RUBINOV, K., ROSCULETE, L., MITRA, T., AND ROYCHOUDHURY, A. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering* (2016), ACM, pp. 923–934.

[45] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[46] SALWAN, J. ROPgadget - Gadgets Finder and Auto-Roper. http://shell-storm.org/project/ROPgadget/, 2011.

[47] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communications Security* (2007), pp. 552–561.

[48] STM32479I-EVAL. http://www.st.com/resource/en/user_manual/dm00219329.pdf.

[49] STM32F4-Discovery. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00037955.pdf.

[50] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP'03: Symposium on Operating Systems Principles* (1993).

[51] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 79–93.

[52] ZELLWEGER, G., GERBER, S., KOURTIS, K., AND ROSCOE, T. Decoupling cores, kernels, and operating systems. In *OSDIUSENIX Symp. on Operating Systems Design and Implementation* (2014), vol. 14, pp. 17–31.

[53] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symp. on Security and Privacy* (2013), IEEE, pp. 559–573.

[54] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *USENIX Security Symp.* (2013), pp. 337–352.

[55] ZHAO, L., LI, G., DE SUTTER, B., AND REGEHR, J. Armor: fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on* (2011), IEEE, pp. 289–298.

[56] ZHOU, Y., WANG, X., CHEN, Y., AND WANG, Z. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 558–569.

[57] ZHOU, Z., YU, M., AND GLIGOR, V. D. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Symp. on Security and Privacy* (2014), IEEE, pp. 308–323.