2017 IEEE Symposium on Security and Privacy

Protecting Bare-metal Embedded Systems With Privilege Overlays

Abraham A. Clements*, Naif Saleh Almakhdhub†, Khaled S. Saab‡, Prashast Srivastava†,
Jinkyu Koo†, Saurabh Bagchi†, Mathias Payer†
*Purdue University and Sandia National Laboratories, clemen19@purdue.edu
†Purdue University, {nalmakhd, srivas41, kooj, sbagchi}@purdue.edu, mathias.payer@nebelwelt.net
‡Georgia Institute of Technology, ksaab3@gatech.edu

***Abstract*—Embedded systems are ubiquitous in every aspect of modern life. As the Internet of Thing expands, our dependence on these systems increases. Many of these interconnected systems are and will be low cost bare-metal systems, executing without an operating system. Bare-metal systems rarely employ any security protection mechanisms and their development assumptions (unrestricted access to all memory and instructions), and constraints (runtime, energy, and memory) makes applying protections challenging.**

To address these challenges we present EPOXY, an LLVM-based embedded compiler. We apply a novel technique, called privilege overlaying, wherein operations requiring privileged execution are identified and only these operations execute in privileged mode. This provides the foundation on which code-integrity, adapted control-flow hijacking defenses, and protections for sensitive IO are applied. We also design fine-grained randomization schemes, that work within the constraints of bare-metal systems to provide further protection against control-flow and data corruption attacks.

These defenses prevent code injection attacks and ROP attacks from scaling across large sets of devices. We evaluate the performance of our combined defense mechanisms for a suite of 75 benchmarks and 3 real-world IoT applications. Our results for the application case studies show that EPOXY has, on average, a 1.8% increase in execution time and a 0.5% increase in energy usage.

I. Introduction

Embedded devices are ubiquitous. With more than 9 billion embedded processors in use today, the number of devices has surpassed the number of humans. With the rise of the "Internet of Things", the number of embedded devices and their connectivity is exploding. These *"things"* include Amazon's Dash button, utility smart meters, smart locks, and smart TVs. Many of these devices are low cost with software running directly on the hardware, known as "bare-metal systems". In such systems, the application runs as privileged low-level software with direct access to the processor and peripherals, without going through intervening operating system software layers. These bare-metal systems satisfy strict runtime guarantees on extremely constrained hardware platforms with few KBs of memory, few MBs of Flash, and low CPU speed to minimize power and cost constraints.

With increasing network connectivity ensuring the security of these systems is critical [21, 51]. In 2016, hijacked smart devices like CCTV cameras and digital video recorders launched the largest distributed denial of service (DDoS) attack to date [39]. The criticality of security for embedded systems extends beyond smart things. Micro-controllers executing bare-metal software have been embedded so deeply into systems that their existence is often overlooked, *e.g.*, in network cards [26], hard drive controllers [57], and SD memory cards [17]. We rely on these systems to provide secure and reliable computation, communication, and data storage. Yet, they are built with security paradigms that have been obsolete for several decades.

Embedded systems largely lack protection against code injection, control-flow hijack, and data corruption attacks. Desktop systems, as surveyed in [53], employ many defenses against these attacks such as: Data Execution Prevention (DEP), stack protections (*e.g.*, stack canaries [22], separate return stacks [31], and SafeStack [40]), diversification [49, 41], ASLR, Control-Flow Integrity [9, 18], or Code-Pointer Integrity (CPI) [40]. Consequently, attacks on desktop-class systems became harder and often highly program dependent.

Achieving known security properties from desktop systems on embedded systems poses fundamental design challenges. *First*, a single program is responsible for hardware configuration, inputs, outputs, and application logic. Thus, the program must be allowed to access all hardware resources and to execute all instructions (*e.g.*, configuring memory permissions). This causes a fundamental tension with best security practices which require restricting access to some resources. *Second*, bare-metal systems have strict constraints on runtime, energy usage, and memory usage. This requires all protections to be lightweight across these dimensions. *Third*, embedded systems are purpose-built devices. As such, they have application-specific security needs. For example, an IO register on one system may unlock a lock while on a different system, it may control an LED used for debugging. Clearly the former is a security-sensitive operation while the latter is not. Such application-specific requirements should be supported in a manner that does not require the developer to make intrusive changes within her application code. *Combined, these challenges have meant that security protection for code injection, control-flow hijack, and data corruption attacks are simply left out from bare-metal systems.*

As an illustrative example, consider the application of DEP

DOI 10.1109/SP.2017.37

to bare-metal systems. DEP, which enforces $W \oplus X$ on all memory regions, is applied on desktops using a Memory Management Unit (MMU), which is not present on micro-controllers. However, many modern micro-controllers have a peripheral called the Memory Protection Unit (MPU) that can enforce read, write, and execute permissions on regions of the physical memory. At first glance, it may appear that DEP can be achieved in a straightforward manner through the use of the MPU. Unfortunately, we find that this is not the case: the MPU protection can be easily disabled, because there is no isolation of privileges. Thus, a vulnerability anywhere in the program can write the MPU's control register to disable it. A testimony to the challenges of correctly using an MPU are the struggles existing embedded OSs have in using it for security protection, even for well-known protections such as DEP. FreeRTOS [1], a popular operating system for low-end micro-controllers, leaves its stacks and RAM to be writable *and* executable. By FreeRTOS's own admission, the MPU port is seldom used and is not well maintained [3]. This was evidenced by multiple releases in 2016 where MPU support did not even compile [8, 2].

To address all of these challenges, we developed EPOXY (Embedded Privilege Overlay on X hardware with Y software), a compiler that brings both generic and system-specific protections to bare-metal applications. This compiler adds additional passes to a traditional LLVM cross-compilation flow, as shown in Figure 1. These passes add protection against code injection, control-flow hijack and data corruption attacks, and direct manipulation of IO. Central to our design is a lightweight *privilege overlay*, which solves the dichotomy of allowing the program developer to assume access to all instructions and memory but restrict access at runtime. To do this, EPOXY reduces execution privileges of the entire application. Then, using static analysis, only instructions requiring elevated privileges are added to the privilege overlay to enable privileges just prior to their execution. EPOXY draws its inputs from a security configuration file, thus decoupling the implementation of security decisions from application design and achieves all the security protections without any application code modification. Combined, these protections provide application-specific security for bare-metal systems that are essential on modern computers.

In adapting fine-grained diversification techniques [41], EPOXY leverages unique aspects of bare-metal systems, specifically all memory is dedicated to a single application and the maximum memory requirements are determined a priori. This enables the amount of unused memory to be calculated and used to increase diversification entropy. EPOXY then adapts the protection of SafeStack [40], enabling strong stack protection within the constraints of bare-metal systems.

Our prototype implemenation of EPOXY supports the ARMv7-M architecture, which includes the popular Cortex-M3, Cortex-M4, and Cortex-M7 micro-controllers. Our techniques are general and should be applicable to any micro-controller that supports at least two modes of execution (privileged and unprivileged) and has an MPU. We evaluate

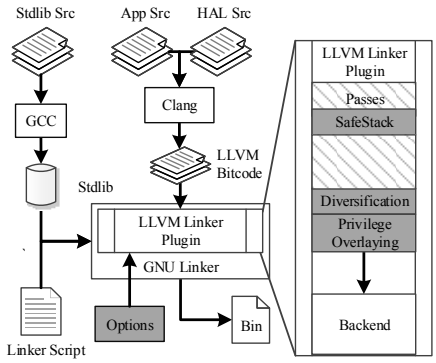


Fig. 1. The compilation work flow for an application using EPOXY. Our modifications are shown in shaded regions.

EPOXY on 75 benchmark applications and three representative IoT applications that each stress different sub-systems. Our performance results for execution time, power usage, and memory usage show that our techniques work within the constraints of bare-metal applications. Overheads for the benchmarks average 1.6% for runtime and 1.1% for energy. For the IoT applications, the average overhead is 1.8% for runtime, and 0.5% for energy. We evaluate the effectiveness of our diversification techniques, using a Return Oriented Programming (ROP) compiler [52] that finds ROP-based exploits. For our three IoT applications, using 1,000 different binaries of each, no gadget survives across more than 107 binaries. This implies that an adversary cannot reverse engineer a single binary and create a ROP chain with a single gadget that scales beyond a small fraction of devices.

In summary, this work: (1) identifies the essential components needed to apply proven security techniques to bare-metal systems; (2) implements them as a transparent runtime privilege overlay, without modifying existing source code; (3) provides state-of-the-art protections (stack protections and diversification of code and data regions) for bare-metal systems within the strict requirements of run-time, memory size, and power usage; (4) demonstrates that these techniques are effective from a security standpoint on bare-metal systems. Simply put, EPOXY brings bare-metal application security forward several decades and applies protections essential for today's connected systems.

II. Threat Model and Platform Assumptions

We assume a remote attacker with knowledge of a generic memory corruption vulnerability, *i.e.*, the application running on the embedded system itself is buggy but not malicious. The goal of the attacker is to either achieve code execution (*e.g.*, injecting her own code, reusing existing code through ROP or performing Data-oriented Programming [37]), corrupt specific data, or directly manipulate security-critical outputs of a system by sending data to specific IO pins. We assume the attacker exploits a write-what-where vulnerability, *i.e.*, one which allows the attacker to write any data to any memory location that she wants. The attacker may have obtained the

vulnerability through a variety of means, *e.g.*, source code analysis, or reverse engineering the binary that runs on a different device and identifying security flaws in it.

We also assume that the attacker does not have access to the specific instance of the (diversified) firmware running on the target device. Our applied defenses provide foundational protections, which are complementary to and assumed by, many modern defenses such as, the memory disclosure prevention work by Braden *et. al.* [15]. We do not protect against attacks that replace the existing firmware with a compromised firmware. Orthogonal techniques such as code signing should be used to prevent this type of attack.

We make the following assumptions about the target system. First, it is running a single bare-metal application, which utilizes a single stack and has no restrictions on the memory addresses, peripherals, or registers that it can access or instructions that it can execute. This is the standard mode of execution of applications on bare-metal systems, *e.g.*, is the case with every single benchmark application and IoT application that we use in the evaluation and that we surveyed from the vendors of the ARM-equipped boards. Second, we require the micro-controller to support at least two execution privilege levels, and have a means to enforce access controls on memory for these privilege levels. These access controls include marking regions of memory as read, write, and/or execute. Typically, an MPU provides this capability on a micro-controller. We looked at over 100 Cortex-M3, M4, and M7 series micro-controllers from ARM and an MPU was present on all but one. Micro-controllers from other vendors, such as AVR32 from Atmel, also have an MPU.

III. Architecture Background Information

This section presents architecture information that is needed to understand the attack vectors and the defense mechanisms in EPOXY. Bare-metal systems have low level access to hardware; this enables an attacker, with a write-what-where vulnerability, to manipulate the system in ways that are unavailable to applications on desktop systems. Defense strategies must consider these attack avenues, and the constraints of hardware available to mitigate threats. For specificity, we focus on the ARMv7-M architecture which is implemented in ARM Cortex-M(3,4,7) micro-controllers. The general techniques are applicable to other architectures subject to the assumptions laid out in Section II. We present key details of the ARMv7-M architecture, full details are in the ARMv7-M Architecture Reference Manual [11].

A. Memory Map

In our threat model, the attacker has a write-what-where vulnerability that can be used to write to any memory address; therefore, it is essential to understand the memory layout of the system. Note that these systems use a single, unified memory space. A representative memory map illustrating the different memory regions is shown in Figure 2. At the very bottom of memory is a region of aliased memory. When an access is made to the aliased region, the access is fulfilled by accessing

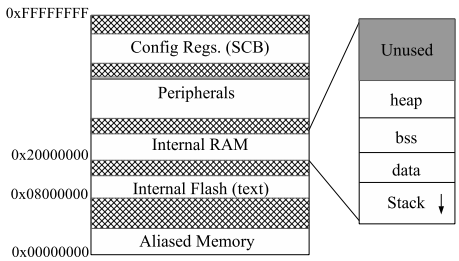


Fig. 2. An example memory map showing the regions of memory commonly available on an ARMv7-M architecture micro-controller. Note the cross hatched areas have an address but no memory.

physical memory that is aliased, which could be in the Internal RAM, Internal Flash, or External Memory. The alias itself is specified through a hardware configuration register. Thus, memory mapped by the aliased region is addressable using two addresses: its default address (*e.g.*, the address of Internal RAM, Internal Flash, or External Memory) and address of the aliased region. This implies that a defender has to configure identical permissions for the aliased memory region and the actual memory region that it points to. A common peripheral (usually a memory controller) contains a memory-mapped register that sets the physical memory addressed by the aliased region. A defender must protect both the register that controls which memory is aliased, in addition to the physical and aliased memory locations.

Moving up the address space we come to Internal Flash, this is Flash memory that is located inside the micro-controller. On ARMv7-M devices it ranges in size from a couple KB to a couple MB. The program code and read only data are usually stored here. If no permissions are enforced, an attacker may directly manipulate code[1]. Address space layout randomization is not applied in practice and the same binary is loaded on all devices, which enables code reuse attacks like ROP. Above the Flash is RAM which holds the heap, stack, and global data (initialized data and uninitialized bss sections). Common sizes range from 1KB to a couple hundred KB and it is usually smaller than the Flash. By default this area is read, write, and execute-enabled, making it vulnerable to code injection attacks. Additionally, the stack employs no protection and thus is vulnerable to stack smashing attacks which can overwrite return addresses and hijack the control flow of the application.

Located above the RAM are the peripherals. This area is sparsely populated and consists of fixed addresses which control hardware peripherals. Peripherals include: General Purpose Input and Output (GPIO), serial communication (UARTS), Ethernet controllers, cryptography accelerators, and many others. Each peripheral is configured, and used by reading and writing to specific memory addresses called memory-mapped registers. For example, a smart lock application will

[1]In Flash a 1 may be changed to a 0 without erasing an entire block, parity checks are also common to detect single bit flips. This restricts the changes that can directly be made to code; however, a wily attacker may still be able to manipulate the code in a malicious way.

use an output pin of the micro-controller to actuate its locking mechanism. In software this will show up as a write to a fixed address. An adversary can directly open the lock by writing to the GPIO register using a write-what-where vulnerability, bypassing any authentication mechanism in the application.

The second region from the top is reserved for external memory and co-processors. This may include things like external RAM or Flash. However, on many small embedded systems nothing is present in this area. If used, it is sparsely populated and the opportunities presented to an attacker are system and program specific. The final area is the System Control Block (SCB). This is a set of memory-mapped registers defined by ARM and present in every ARMv7-M microcontroller. It controls the MPU configuration, interrupt vector location, system reset, and interrupt priorities. Since the SCB contains the MPU configuration registers, an attacker can disable the MPU simply by writing a 0 to the lowest bit of the *MPU_CTRL* register located at address 0xE000ED94. Similarly, the location of the interrupt vector table is set by writing the *VTOR* register at 0xE000ED08. These indicate that the SCB region is critical from a security standpoint.

B. Execution Privileges Modes

Like their x86 counterparts, ARMv7-M processors can execute in different privilege modes. However, they only support two modes: privileged and unprivileged. In the current default mode of operation, the entire application executes in privileged mode, which means that all privileged instructions and all memory accesses are allowed. Thus, we cannot indiscriminately reduce the privilege level of the application, for fear of breaking the application's functionality. Once privileges are reduced the only way to elevate privileges is through an exception. All exceptions execute in privileged mode and software can invoke an exception by executing an SVC (for "supervisor call") instruction. This same mechanism is used to create a system call in a traditional OS.

C. Memory Protection Unit

ARMv7-M devices have a Memory Protection Unit or MPU which can be used to set read, write, or execute permissions on regions of the physical memory. The MPU is similar to an MMU, but it does not provide virtual memory addressing. In effect, the MPU adds an access control layer over the physical memory but memory is still addressed by its physical addresses. The MPU defines read, write, and execute privileges for both privileged and unprivileged modes. It also enables making regions of memory non executable ("execute never" in ARM's terminology). It supports setting up to 8 regions, numbered from 0 to 7, with the following restrictions: (1) A region's size can be from 32 Bytes to 4 GBytes, in powers of two; (2) Each region must be size-aligned (*e.g.*, if the region is 16KB, it must start on a multiple of 16KB); (3) If there is a conflict of permissions (through overlapping regions), then the higher numbered region's permissions take effect. Figure 3 illustrates how memory permissions are applied.

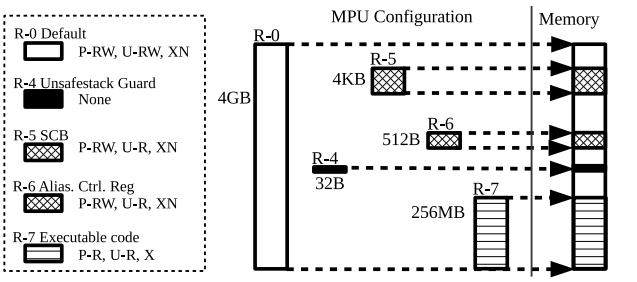


Fig. 3. Diagram illustrating how the protection regions (R-x) defined in the MPU by EPOXY are applied to memory. Legend shows permissions and purpose of each region. Note regions R1-R3 (not shown) are developer defined.

For the remainder of this paper we will use the following notations to describe permissions for a memory region: *(P-$R^?W^?$,U-$R^?W^?$,X|$-^?$)* which encodes read and write permissions for privileged mode (P), unprivileged mode (U), and execution permission for both privileged and unprivileged mode. For example, the tuple *(P-RW,U-R,X)* encodes a region as executable, read-write for privileged mode and executable, read-only access for unprivileged mode. Note, execute permissions are set for both privileged and unprivileged mode. For code to be executed, read access must be granted. Thus, unprivileged code can be prevented from executing a region by removing read access to it.

D. Background Summary

Current bare-metal system design exposes a large attack surface—memory corruption, code injection, control-flow hijack attachs, writing to security-critical but system-specific IO, and modification of registers crucial for system operation such as the SCB and MPU configuration. Execution privilege modes and the MPU provide the hardware foundation that can be used to develop techniques that will reduce this vast attack surface. However, the development assumption that all instructions and all memory locations are accessible is in direct conflict with the security requirements, as some instructions and memory accesses can exploit the attack surface and need to be restricted. Next we present the design of our solution EPOXY, which resolves this tension by using privilege overlays, along with various diversification techniques to remove the attack surface.

IV. Design

EPOXY's goal is to apply system specific protections to bare-metal applications. This requires meeting several requirements: (1) Protections must be flexible as protected areas vary from system to system; (2) The compiler must enable the enforcement of policies that protect against malicious code injection, code reuse attacks, global data corruption, and direct manipulation of IO; (3) Enforcement of the policies must satisfy the non-functional constraints—runtime, energy usage, and memory usage should not be significantly higher

than in the baseline insecure execution. (4) The protections should not cause the application developers to make changes to their development workflow and ideally would involve no application code changes.

EPOXY's design utilizes four components to apply protections to bare-metal systems, while achieving the above four goals. They are: (1) access controls which limit the use of specific instructions and accesses to sensitive memory locations, (2) our novel privilege overlay which imposes the access control on the unmodified application, (3) an adapted SafeStack, and (4) diversification techniques which utilize all available memory.

A. Access Controls

Access controls are used to protect against code injection attacks and defend against direct manipulation of IO. Access controls specify the read, write, and execute permissions for each memory region and the instructions which can be executed for a given execution mode. As described in Section III, modern micro-controllers contain an MPU and multiple execution modes. These are designed to enable DEP and to restrict access to specific memory locations. We utilize the MPU and multiple execution modes to enforce access controls in our design. Using this available hardware, rather than using a software only approach, helps minimize the impact on runtime, energy consumption, and memory usage. On our target architecture, IO is handled through memory-mapped registers as well and thus, the MPU can be used to restrict access to sensitive IO. The counter argument to the use of the MPU is that it imposes restrictions—how many memory regions can be configured (8 in our chosen ARM architecture) and how large each region needs to be and how it should be aligned (Section III-C). However, we still choose to use the MPU and this explains in part the low overhead that EPOXY incurs (Table II). While the MPU and the processor execution modes can enforce access controls at runtime they must be properly configured to enable robust protection. We first identify the proper access controls and how to enforce them. We then use the compiler to generate the needed hardware configuration to enforce access controls at runtime. Attempts to access disallowed locations trap to a fault handler. The action the fault handler takes is application specific, *e.g.*, halting the system, which provides the strongest protects as it prevents repeated attack attempts.

The required access controls and mechanisms to enforce them can be divided into two parts: architecture dependent and system specific. Architecture-dependent access controls: All systems using a specific architecture (*e.g.*, ARMv7-M) have a shared set of required access controls. They must restrict access to instructions and memory-mapped registers that can undermine the security of the system. The instructions that require execution in privileged mode are specified in the processor architecture and are typically those that change special-purpose registers, such as the program status register (the `MSR` and `CPS` instructions). Access to these instructions is limited by executing the application by default in unprivileged mode. Memory-mapped registers, such as the MPU configuration registers, and interrupt vector offset register, are common to an architecture and must be protected. In our design, this is done by configuring the MPU to only allow access to these regions (registers) from the privileged mode.

System-specific access controls: These are composed of setting $W \oplus X$ on code and data, protection of the alias control register, and protecting any sensitive IO. $W \oplus X$ should be applied to every system; however, the locations of code and data change from system to system, making the required configuration to enforce it system specific. For example, each micro-controller has different amounts of memory and a developer may place code and data in different regions, depending on her requirements. The peripheral that controls the aliased memory is also system specific and needs protection and thus, access to it should be set for the privileged mode only. Last, what IO is sensitive varies from system to system and only the subset of IO that is sensitive need be restricted to the privileged mode.

To simplify the implementation of the correct access controls, our compiler generates the necessary system configuration automatically. At the linking stage, our compiler extracts information (location, size, and permissions) for the code region and the data region. In addition, the developer provides on a per-application basis information about the location and size of the alias control register and what IO is sensitive. The compiler then uses this information, along with the architecture-specific access controls, to generate the MPU configuration. The MPU configuration requires writing the correct bits to specific registers to enforce the access controls. Our compiler pass adds code to system startup to configure the MPU (Figure 3 and Table I). The startup code thus drops the privileges of the application that is about to execute, causing it to start execution in unprivileged mode.

B. Privilege Overlay

We maintain the developer's assumption of access to all instructions and memory locations by using a technique that we call, *privilege overlay*. This technique, identifies all instructions and memory accesses which are restricted by the access controls—referred to as **restricted operations**—and elevates just these instructions. Conceptually, this is like overlaying the original program with a mask which elevates just those instructions which require privileged mode. In some ways, this privilege overlaying is similar to an application making an operating system call and transitioning from unprivileged mode to privileged mode. However, here, instead of being a fixed set of calls which operate in the operating system's context, it creates a minimal set of instructions (loads and stores from and to sensitive locations and two specific instructions) that execute in their original context (the only context used in a bare-metal application execution) after being given permissions to perform the restricted operation. By elevating just those instructions which perform restricted operations through the privilege overlay, we simplify the development

process and by carefully selecting the restricted operations, we limit the power of a write-what-where vulnerability.

Privilege overlaying requires two mechanisms: A mechanism to elevate privileges for just the restricted operations and a mechanism to identify all the restricted operations. Architectures employing multiple execution modes provide a mechanism for requesting the execution of higher level software. On ARM, this is the SVC instruction which causes an exception handler to be invoked. This handler checks if the call came from an authorized location, and if so, it elevates the execution mode to the privileged mode and returns to the original context. If it was not from an authorized location, then it passes the request on to the original handler without elevating the privilege, *i.e.*, it denies the request silently. The compiler identifies each restricted operation and prepends it with a call to the SVC handler and, immediately after the restricted operation, adds instructions that drop the execution privileges. Thus, each restricted operation executes in privileged mode and then immediately returns to unprivileged mode.

The restrictions in the way MPU configuration can be specified, creates challenges for EPOXY. The MPU is restricted to protecting blocks of memory of size at least 32 Bytes, and sometimes these blocks include both memory-mapped registers that must be protected to ensure system integrity, and those which need to be accessed for correct functionality. For example, the Vector Table Offset Register (VTOR) and the Application Interrupt and Reset Control Register (AIRCR) are immediately adjacent to each other in one 32 Byte region. The VTOR is used to point to the location of the interrupt vector table and is thus a security critical register, while the AIRCR is used (among other things) for the software running on the device to request a system reset (say, to reload a new firmware image) and is thus not security critical. There is no way to set permissions on the VTOR without also applying the same permissions to the AIRCR. EPOXY overcomes this restriction by adding accesses to the AIRCR to the privilege overlay, thus elevating accesses whenever the AIRCR is being accessed.

C. Identifying Restricted Operations

To identify restricted operations we utilize static analysis and optionally, source code annotations by the developer. Using static analysis enables the compiler to identify many of the restricted operations, reducing the burden on the developer. We use two analyses to identify restricted operations; one for restricted instructions and a second to identify restricted memory accesses. Restricted instructions are defined by the Instruction Set Architecture (ISA) and require execution in privileged mode. For the ARMv7-M architecture these are the CPS and MSR instructions, each of which controls specific flags in the program status register, such as enabling or disabling interrupt processing. These privileged instructions are identified by string matching during the appropriate LLVM pass. Identifying restricted memory accesses however is more challenging.

An important observation enables EPOXY to identify most restricted accesses. In our case, the memory addresses being accessed are memory-mapped registers. In software, these accesses are reads and writes to fixed addresses. Typically, a Hardware Abstraction Layer (HAL) is used to make these accesses. Our study of HAL's identified three patterns that cover most accesses to these registers. The first pattern uses a macro to directly access a hard-coded address. The second pattern uses a similar macro and a structure to access fixed offsets from a hard-coded address. The last pattern uses a structure pointer set to a hard-coded address. All use a hard-coded address or fixed offsets from them. The use of hard-coded addresses, and fixed offsets from them, are readily identifiable by static analysis.

Our static analysis uses backward slicing to identify these accesses. A backward slice contains all instructions that affect the operands of a particular instruction. This enables identifying the potential values of operands at a particular location in a program. We limit our slices to a single function and examine only the definitions for the address operand of load and store operations. Accesses to sensitive registers are identified by checking if the address being accessed is derived from a constant address. This static analysis captures many of the restricted memory accesses; however, not all accesses can be statically identified and manual annotations (likely by the developer) are required in these cases. Note that we observed few annotations in practice and most are generic per hardware platform, i.e., they can be provided by the manufacturer. This primarily occurs when memory-mapped registers are used as arguments in function calls or when aliasing of memory-mapped registers occurs. Aliasing occurs when the register is not directly referenced, but is assigned to a pointer, and multiple copies of that pointer are made so that the register is now accessible via many different pointers. These point to two limitations of our current static analysis. Our backward slicing is limited to a single function and with some bounded engineering effort, we can expand it to perform inter-procedural analysis. To overcome the second limitation though requires precise alias analysis, which is undecidable in the general case [50]. However, embedded programs—and specifically access to memory mapped registers—are constrained in their program structures reducing the concern of aliasing in this domain.

D. Modified SafeStack

EPOXY defends against control-flow hijacking attack by employing SafeStack [40], modified to bare-metal systems. SafeStack is a protection mechanism that uses static analysis to move local variables which may be used in an unsafe manner to a separate *unsafestack*. A variable is unsafe if it may access memory out-of-bounds or if it escapes the current function. For example, if a supplied parameter is used as the index of an array access, the array will be placed on the *unsafestack*. It utilizes virtual addressing to isolate the *unsafestack* from the rest of the memory. By design, return addresses are always placed on the regular stack because they have to be protected from illegal accesses. SafeStack ensures that illegal accesses

may only happen on items on the *unsafestack*. In addition to its security properties, Safestack has low runtime overhead (generally below 1% [40] §5.2) and a deterministic impact on stack sizes makes it a good fit for bare-metal systems. The deterministic impact means—assuming known maximum bounds for recursion—the maximum size for both the regular and *unsafestack* is fixed and can be determined a priori. Use of recursion without knowing its bounds is bad design for bare-metal systems.

While the low runtime overhead of SafeStack makes it suitable for bare-metal systems, it needs an isolated memory region to be effective. The original technique, deployed on Intel architectures, relied on hardware support for isolation (either segmentation or virtual memory) to ensure low overhead. For example, it made the safe region accessible through a dedicated segment register, which is otherwise unused, and configured limits for all other segment registers to make the region inaccessible through them (on x86). *Such hardware segment registers and hardware protection are not available in embedded architectures.* The alternate pure software mechanism based on Software Fault Isolation [56] would be too expensive for our embedded applications because it requires that all memory operations in a program are masked. While on some architectures with a large amount of (virtual) memory, this instrumentation can be lightweight (e.g., a single `and` operation if the safe region occupies a linear part of the address space – encoded in a mask, resulting in about 5% overhead), here masking is unlikely to work because the safe region will occupy a smaller and unaligned part of the scarce RAM memory.

Therefore, to apply the SafeStack principle to bare-metal systems, we place the *unsafestack* at the top of the RAM, and make the stack grow up, as shown in Figure 4a. We then place a guard between the *unsafestack* and the other regions in RAM, shown as the black region in the figure. This follows best practices for embedded systems to always grow a stack away from other memory regions. The guard is created as part of the MPU configurations generated by the compiler. The guard region is inaccessible to *both* privileged *and* unprivileged code (*i.e.*, privileges are *(P-,W-,XN)*). Any overflow on the *unsafestack* will cause a fault either by accessing beyond the bounds of memory, or trying to access the guard region. It also prevents traditional stack smashing attacks because any local variable that can be overflown will be placed on the *unsafestack* while return addresses are placed on the regular stack. Our design for the first time provides strong stack protection on bare-metal embedded systems.

V. Implementation

A. Access Controls

We developed a prototype implementation of EPOXY, building on LLVM 3.9 [42]. In our implementation, access controls are specified using a template. The template consists of a set of regions that map to MPU region configurations (see Section III-C for the configuration details). Due to current hardware restrictions, a maximum of 8 regions are supported.

TABLE I
THE MPU CONFIGURATION USED FOR EPOXY. FOR OVERLAPPING REGIONS THE HIGHEST NUMBERED REGION (R) TAKES EFFECT.

R	Permissions	Start Addr	Size	Protects
0	P-RW,U-RW,XN	0x00000000	4GB	Default
4	None	Varies	32B	*unsafestack* Guard
5	P-RW,U-R,XN	0xE000E000	4KB	SCB
6	P-RW,U-R,XN	0x40013800	512B	Alias. Ctrl. Reg
7	P-R,U-R,X	0x00000000	256MB	Executable Code

Our basis template uses five regions as shown in Table I. Region 0 encodes default permissions. Using region 0 ensures all other regions override these permissions. We then use the highest regions and work down to assign permissions to ensure that the appropriate permissions are enforced. Region 7 is used to enforce $W \oplus X$ on executable memory. This region covers both the executable memory and its aliased addresses starting at address 0. The three remaining regions (4-6) can be defined in any order and protect the SCB, alias control register, and the *unsafestack* guard.

The template can be modified to accommodate system specific requirements, *e.g.*, changing the start address and size of a particular region. For example, the two micro-controllers used for evaluation place the alias control register at different physical addresses. Thus, we modified the start address and size for each micro-controller. Regions 1-3 are unused and can be used to protect sensitive IO that is application specific. To do this, the start address and size cover the peripheral and permissions are set to *(P-RW,U-RW,XN)*. The addresses for all peripherals are given in micro-controller documentation provided by the vendor. The use of the template enables system specific access controls to be placed on the system. It also decouples the development of access control mechanisms and application logic.

We implemented a pass in LLVM that generates code to configure the MPU based on the template. The code writes the appropriate values to the MPU configuration registers to enforce the access controls given in the template, and then reduces execution privileges. The code is called at the very beginning of *main*. Thus all of *main* and the rest of the program executes with reduced privileges.

B. Privilege Overlays

Privileged overlay mechanisms (*i.e.*, privilege elevation and restricted operation identification) are implemented using an LLVM pass. To elevate privileges two components are used. They are a privilege requester and a request handler. Requests are made to the handler by adding code which performs the operations around restricted operations, as shown in Algorithm 1. This code saves the execution state and executes a SVC (SVC FE) to elevate privileges. The selected instructions are then executed in privileged mode, followed by a code sequence that drops privileges by setting the zero bit in the control register. Note that this sequence of instructions can safely be executed as part of an interrupt handler routine as interrupts execute with privileges and, in that mode, the CPU ignores both the SVC instruction and the write to the control register.

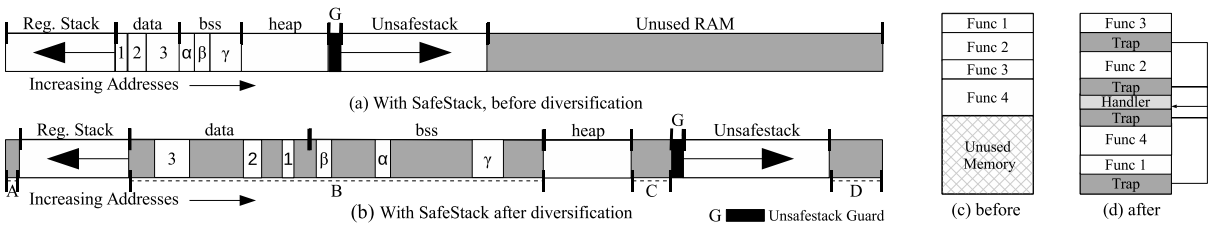


Fig. 4. Diagrams showing how diversification is applied. (a) Shows the RAM layout with SafeStack applied before diversification techniques are applied. (b) Shows RAM the layout after diversification is applied. Note that unused memory (gray) is dispersed throughout RAM, the order of variables within the data section (denoted 1-7) and bss section (greek letters) are randomized. Regions A, B, C, and D are random sizes, and G is the *unsafestack* guard region. (c) Layout of functions before protection; (d) Layout of functions after trapping and randomizing function order.

Algorithm 1 Procedure used to request elevated privileges

```
1: procedure REQUEST PRIVILEGED EXECUTION
2:     Save Register and Flags State
3:     if In Unprivileged Mode then
4:         Execute SVC FE (Elevates Privileges)
5:     end if
6:     Restore Register and Flags
7:     Execute Restricted Operation
8:     Set Bit 0 of Control Reg (Reduces Privileges)
9: end procedure
```

Algorithm 2 Request handler for elevating privileges

```
1: procedure HANDLE PRIVILEGE REQUEST
2:     Save Process State
3:     if Interrupt Source == SVC FE then
4:         Clear bit 0 of Control Reg (Elevates Privileges)
5:         Return
6:     else
7:         Restore State
8:         Call Original Interrupt Handler
9:     end if
10: end procedure
```

The request handler intercepts three interrupt service routines and implements the logic shown in Algorithm 2. The handler stores register state (R0-R3 and LR – the remaining registers are not used) and checks that the caller is an SVC FE instruction. Authenticating the call site ensures that only requests from legitimate locations are allowed. Due to $W \oplus X$, no illegal SVC FE instruction can be injected. If the interrupt was caused by something other than the SVC FE instruction the original interrupt handler is called.

The request handler is injected by the compiler by intercepting three interrupt handlers. These are: the SVC handler, the Hard Fault handler, and the Non Maskable Interrupt handler. Note that executing an SVC instruction causes an interrupt. When interrupts are disabled the SVC results in a Hard Fault. Similarly, when the Fault Mask is set all interrupt handlers except the Non-Maskable Interrupt handler are disabled. If an SVC instruction is executed when the fault mask is set it causes a Non-Maskable Interrupt. Enabling and disabling both interrupts and faults are privileged operations, thus all three interrupt sources need to be intercepted by the request handler.

Privileged requests are injected for every identified restricted operation. The static analyses used to identify restricted operations are implemented in the same LLVM pass. It adds privilege elevation request to all *CPS* instructions, and all *MSR* instructions that use a register besides the *APSR* registers. These instructions require execution in privileged mode. To detect loads and stores from constant addresses we use LLVM's *use-def chains* to get the back slice for each load and store. If the pointer operand can be resolved to a constant address it is checked against the access controls applied in the MPU. If the MPU's configuration restricts that access a privilege elevation request is added around the operation. This identifies many of the restricted operations. Annotations can be used to identify additional restricted operations.

C. SafeStack and Diversification

The SafeStack in EPOXY extends and modifies the SafeStack implemented in LLVM 3.9. Our changes enable support for the ARMv7-M architecture, change the stack to grow up, and use a global variable to store the *unsafestack* pointer. Stack offsets are applied with global data randomization. Global data randomization is applied using a compiler pass. It takes the amount of unused RAM as a parameter which is then randomly split into five groups. These groups specify how much memory can be used in each of the following regions: stack offset, data region, bss region, *unsafestack* offset, and unused. The number of bytes added to each section is a multiple of four to preserve alignment of variables on word boundaries. The data and bss region diversity is increased by adding dummy variables to each region. Note that adding dummy variables to the data regions increases the Flash used because the initial values for the data section are stored as an array in the Flash and copied to RAM at reset. However, Flash capacity on a micro-controller is usually several times larger than the RAM capacity and thus, this is less of a concern. Further an option can be used to restrict the amount of memory for dummy variables in the data section. Dummy variables in the bss do not increase the amount of Flash used.

Another LLVM pass is used to randomize the function order. This pass takes the amount of memory that can be dispersed throughout the text section. It then disperses this memory between the function by adding trap functions to the global function list. The global function list is then randomized, and the linker lays out the functions in the shuffled order in

the final binary. A trap function is a small function which, if executed, jumps to a fault handler. These traps are never executed in a benign execution and thus incur no runtime overhead but detect unexpected execution.

VI. Evaluation

We evaluate the performance of EPOXY with respect to the design goals, both in terms of security and resource overhead. We first evaluate the impact on runtime and energy using a set of benchmarks. We then use three real-world IoT applications to understand the effects on runtime, energy consumption, and memory usage. Next, we present an evaluation of the effectiveness of the security mechanisms applied in EPOXY. This includes an evaluation of the effectiveness of diversification to defeat ROP-based code execution attacks and discussion of the available entropy. We complete our evaluation by comparing our solution to FreeRTOS with respect to the three IoT applications.

Several different kinds of binaries are evaluated for each program using different configurations of EPOXY these are: (1) unmodified baseline, (2) privilege overlays (i.e., applies privilege overlaying to allow the access controls to protect system registers and apply $W \oplus X$.), (3) SafeStack only, and (4) fully protected variants that apply privileged overlaying, SafeStack, and software diversity. We create multiple variants of a program (20 is the default) by providing EPOXY a unique diversification seed. All binaries were compiled using link time optimization at the *O2* level.

We used two different development boards for our experiments the STM32F4Discovery board [6] and the STM32F479I-Eval [5] board. Power and runtime were measured using a logic analyzer sampling execution time at 100Mhz. Each application triggers a pin at the beginning and at the end of its execution event. A current sensor with power resolution of 0.5 μW was attached in series with the micro-controller's power supply enabling only the power used by the micro-controller to be measured. The analog power samples were taken at 125 KHz, and integrated over the execution time to obtain the energy consumption.

A. Benchmark Performance Evaluation

To measure the effects of our techniques on runtime and energy we use the BEEBs benchmarks [47]. The BEEBs' benchmarks are a collection of applications from MiBench [34], WCET [33] and DSPstone [60] benchmarks. They were designed and selected to measure execution performance and energy consumption under a variety of computational loads. We selected the 75 (out of 86) BEEBs' benchmarks that execute for longer than 50,000 clock cycles, and thus, providing a fair comparison to real applications. For reference, our shortest IoT application executes over 800,000 clock cycles. Each is loaded onto the Discovery board and the logic analyzer captures the runtime and energy consumption for 64 iterations of the benchmark for each binary.

Across the 75 benchmarks the average overhead is 1.6% for runtime and 1.1% for energy. The largest increase is on *cover* 14.2% runtime, 17.9% energy and largest decrease on *compress* (-11.7% runtime, -10.2% energy). *ctl_stack* is the only other benchmark that has a change in runtime (13.1%) or energy (15.8%) usage that exceeds $\pm$10%. Table II shows the runtime and energy overheads for the benchmarks executing over 2 million clock cycles. The remaining benchmarks are omitted for space. We find runtime is the biggest factor in energy consumption—the Spearman's rank correlation coefficient is a high 0.8591.

TABLE II
The runtime and energy overheads for the benchmarks executing over 2 million clock cycles. Columns are SafeStack only (SS), privilege overlay only (PO), and all protections of EPOXY applied, averaged across 20 variants (All), and the number of clock cycles each benchmark executed, in millions of clock cycles. Average is for all 75 benchmarks

	% Runtime			%Energy			Clk
Benchmark	SS	PO	All	SS	PO	All	
crc32	0.0	0.0	2.9	-0.1	-0.6	2.5	2.2
sg..insearch	0.0	0.2	-1.0	-0.2	-0.9	0.5	2.2
ndes	2.9	-0.2	1.3	2.4	1.2	3.4	2.4
levenshtein	1.5	0.0	3.0	1.7	0.8	3.8	2.6
sg..quicksort	-2.3	0.0	-1.4	-2.8	-0.5	-0.3	2.7
slre	-1.5	-0.3	5.3	-2.0	-0.3	8.1	2.9
sgl..htable	-0.6	0.0	2.0	-1.0	-0.7	3.4	2.9
sgl..dllist	-0.6	0.0	0.7	0.3	-0.1	2.6	3.7
edn	0.0	-0.1	0.8	1.9	1.5	4.2	3.8
sg..insertsort	-0.3	0.0	1.7	-0.1	-1.6	1.6	3.9
sg..heapsort	0.0	0.0	-0.5	-0.1	1.4	1.9	4.0
sg..queue	-7.3	0.0	-7.3	-4.2	-0.9	-3.4	4.6
sg..listsort	-0.4	0.0	0.7	-0.1	-0.5	2.4	4.9
fft	0.0	0.4	0.4	-0.1	0.6	-0.3	5.1
bubblesort	0.0	0.0	1.7	-0.1	1.0	2.6	6.8
matmult_int	0.0	0.0	1.2	-0.1	-0.4	0.7	6.8
adpcm	0.0	0.1	-0.4	0.1	2.3	0.6	7.3
sglib_rbtree	-0.2	-0.1	2.4	0.1	-0.7	3.7	7.4
mat..float	0.0	0.6	0.7	0.0	0.1	1.2	8.6
frac	1.6	2.0	1.7	2.4	2.8	4.0	9.9
st	0.0	0.1	0.4	-0.9	-0.3	1.2	19.0
huffbench	1.3	0.0	1.5	7.3	1.2	4.5	20.9
fir	-1.0	-1.0	1.7	-2.0	1.5	3.1	21.0
cubic	-0.2	0.2	0.1	0.0	-0.2	0.6	30.1
stb_perlin	0.0	-1.3	0.0	0.0	-3.0	0.4	31.6
mergesort	-0.2	0.5	2.1	-1.0	-0.4	3.1	44.0
qrduino	0.0	0.0	-1.2	-0.1	-0.7	-0.6	46.0
picojpeg	0.0	-0.4	-2.4	0.0	0.0	0.2	54.3
blowfish	-0.4	0.0	-1.3	1.4	-1.3	0.5	56.9
dijkstra	0.0	-0.1	-8.7	-0.1	0.0	-7.3	70.5
rijndael	-1.1	0.0	0.1	-0.6	-0.4	2.0	94.9
sqrt	0.0	2.1	1.4	0.0	1.8	2.1	116.2
whetstone	-0.4	-0.3	0.1	0.8	0.3	1.6	135.5
nbody	1.1	1.1	0.4	0.9	0.9	2.5	139.0
fasta	0.0	0.0	0.4	0.1	0.4	1.2	157.1
wikisort	0.3	0.9	2.1	0.2	0.1	3.0	179.6
lms	0.0	0.1	0.6	-0.1	0.3	0.2	225.2
sha	-3.5	0.0	-3.7	-1.3	-0.2	0.2	392.9
Average	**0.1**	**0.1**	**1.1**	**0.2**	**-0.2**	**2.5**	**26.3**

The impact on execution time can be explained by the application of SafeStack (*e.g.*, *sg..queue* in Table II) and diversification. Modest improvements in execution time were found by the creators of SafeStack ([40] §5.2), the primary cause being improvements in locality. Likewise, our improvements come from moving some variables to the *unsafestack*. These typically tend to be larger variables like arrays. This increases the locality of remaining variables on the regular stack and

enables them to be addressed from offsets to the stack pointer, rather than storing base addresses in registers and using offsets from these. This frees additional registers to store frequently used variables, thus reducing register spilling, and consequent writes and reads to the stack, thereby improving execution time. The impact of the privilege overlay on the running time is minimal because these benchmarks have few restricted operations in them and the setups due to EPOXY (such as MPU configuration) happen in the startup phase which is not measured for calculating the overhead.

Diversification changes execution time in two ways. The first is locality of functions and variables relative to each other. Consider separately the case of a control-flow transfer and a memory load/store. When a control-flow transfer is done (say a branch instruction) and the target is close by, then the target address is created relative to the PC and control flow is transferred to that address (1 instruction). On the other hand, if the target address is farther off, then a register is loaded with the address (2 instructions) and control transferred to the content of the register (1 instruction). Sometimes diversification puts the callee and called function farther apart than in the baseline in which case the more expensive operation is used. In other cases the opposite occurs, enabling less expensive (compared to the baseline) control transfer to be used. Similarly, when a memory load (or store) is done from a far off location, a new register needs to be loaded with the address and then the location accessed (3 instructions), while if it were to a location near an address already in a register, then it can be accessed using an offset from that register as the base address (1 instruction). The dispersed accesses also uses more registers, increasing register pressure.

Another effect of diversification is even more subtle and architecture specific. In our target ARM architecture, when a caller invokes a function, general-purpose registers R0-R3 are assumed to be used and overwritten by the callee function and therefore the compiler does not need to save the values of those registers in the callee context. Thus the compiler gives preference to using R0-R3 when allocating registers. Due to our register randomization this preference is not always followed, and other general purpose registers (R4-R13) are used more often than they are in the baseline case. When R4-R13 are used they first must be saved to, and restored from the stack, decreaseing performance. To partially alleviate this performance hit, EPOXY in its register randomization favors the use of the registers R0-R3 in the callee function through a non-uniform stochastic process, but does not deterministically enforce this. Reassuringly, the net effect from all the instances of the diversification is only a small increase in the runtime—a worst case of 14.7% and an average of 1.1% across all the benchmark applications.

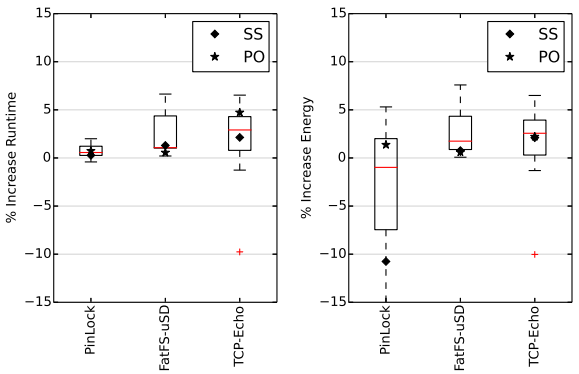
B. Application Performance Evaluation

Benchmarks are useful for determining the impact of our techniques under controlled conditions. To understand the overall effects on realistic applications, we use three representative IoT applications. Our first program, PinLock, simulates a simple IoT device like a door lock. It requests a four digit pin be entered over a serial port. Upon reception the pin is hashed, using SHA1, and compared to a precomputed hash. If the hashes match, an LED is turned on, indicating the system is unlocked. If an incorrect pin is received the user is prompted to try again. In this application the IO is restricted to privileged mode only, thus each time the lock is unlocked, privileged execution must first be obtained. This demonstrates EPOXY's ability to apply application specific access controls. We repeatedly send an incorrect pin followed by the correct pin and measure time between successful unlocks. The baud rate (115,200 max standard rate) of the UART communications is the limiting factor in how fast login attempts are made.

We also use two vendor applications provided with the STM32F479I-Eval board. The FatFS-uSD program implements a FAT file system on a micro-SD card. It creates a file on the SDCard, writes 1KB of data to the file and then reads back the contents and checks that they are the same. We measure the time it takes to write, read and verify the file. The TCP-Echo application implements a TCP/IP stack and listens for a packet on the Ethernet connection. When it receives a packet it echoes it back to the receiver. We measure the time it takes to send and receive 1,000 packets, with requests being sent to the board fast enough to fully saturate the capabilities of the STM32F479I-Eval board (*i.e.*, computation on the board is the limiting factor in how fast packets are sent and received).

For each of the three applications we create the same set of binaries used for the benchmarks: baseline, SafeStack only, privilege overlay only, and 20 variants with all protections of EPOXY. To obtain runtime and energy consumption we average 10 executions of each binary. Percent increase relative to the baseline binary is taken for each binary. The average runtime overhead is 0.7% for PinLock, 2.4% for FatFS-uSD, and 2.1% for TCP-Echo. Figure 5a shows the execution time overheads as a whisker plot. In the worst case among all executions of all applications protected with EPOXY, the runtime overhead is 6.5% occurring on TCP-Echo. Again we see energy consumption is closely related to execution time. Each application's average energy overheads are: -2.9% for PinLock, 2.6% for FatFS-uSD and 1.8% for TCP-Echo. Figure 5b shows the energy consumption overheads, with a noticeable difference: PinLock has a very tight runtime distribution, and a relatively wide energy distribution. This application is IO bound and the application is often waiting to receive a byte over the serial port, due to the slow serial connection, causing the time variation to be hidden. However, the changed instruction mix due to EPOXY still causes variation in energy overhead.

Changes in memory usage are shown in Table III. It shows the averages of increase to code (text section), global data (data and bss sections), and stack usage for the 20 variants of each application. SafeStack, privilege overlaying, and diversification can all affect the code size. SafeStack increases the code size by requiring additional instructions to manage a second stack, privilege overlaying directly injects new code, and as discussed previously diversification can cause the compiler to



(a) (b)

Fig. 5. Box plots showing percent increase in execution time (a) and energy (b) for the three IoT applications. The diamond shows the SafeStack only binary, and the star shows the privilege overlay only binary.

TABLE III
INCREASE IN MEMORY USAGE FOR THE IOT APPLICATIONS FROM APPLYING ALL OF EPOXY'S PROTECTIONS.

App	Text	Global Data	Stack	
			SafeStack	Priv. Over.
PinLock	3,390 (29%)	14.6 (1%)	104 (25%)	0
FatFs-uSD	2,839 (12%)	18.2 (1%)	128 (3%)	36 (1%)
TCP-Echo	3,249 (8%)	7.2 (0%)	128 (29%)	0

emit varying code. In all, we find that all the three applications needed less than 3,390 additional bytes for code. For PinLock (the smallest application) which has a baseline text size of 11,788 bytes, the additional 3,390 bytes would still fit in 16KB Flash, thus the same micro-controller could be used with EPOXY's protections. Impacts on data are caused by SafeStack (4 bytes for the *unsafestack* pointer), and a few bytes added to preserve alignment of variables. The majority of the increase in stack size come from applying SafeStack. It accounts for all the increase in PinLock and 128 bytes in both FatFS-uSD and TCP-Echo. SafeStack increases the stack requirements, because splitting the stack requires memory for the *sum* of the execution paths with the deepest regular stack and the deepest *unsafestack* across all possible execution paths. In comparison, for the baseline, which has a single stack, only memory for the deepest execution path is required. Privilege overlays may also require additional memory—to save and restore state while elevating privileges—but extra memory is only needed when it increases the stack size of the deepest execution path. Thus, additional memory, beyond SafeStack is not needed for PinLock or TCP-Echo.

From the performance and memory usage requirements we find that EPOXY's protections operate within the non-functional constraints of runtime, energy consumption and memory usage. It also greatly reduces the burden on the developer. For all BEEBs benchmarks, FatFS-uSD, and TCP-Echo (77 applications in all), a total of 10 annotations were made. These annotations were all made in ARM's CMSIS library—a C-language Hardware Abstraction Library (HAL) for common ARM components—which is shared across the 77 applications. PinLock required an additional 7 annotations to protect its IO. We envision HAL writers could provide pre-annotated libraries, further reducing the burden on developers. The annotations were all required because offsets were passed as arguments to functions and a store was done by adding the offset to a constant address. Extending our analysis to be inter-procedural will allow the compiler to handle these cases and remove the need for manual annotation. Our compiler elevated 35 (PinLock), 31(FatFs-uSD), and 25 (TCP-Echo) operations on the IoT applications.

C. Security Evaluation

EPOXY meets the design goals for usability and performance, but does it provide useful protection? First, EPOXY enables the application of $W \oplus X$, a proven protection against code injection and is foundational for other protections. Our $W \oplus X$ mechanism also protects against attacks which attempt to bypass or disable $W \oplus X$ by manipulating system registers using a write-what-where vulnerability. EPOXY incorporates an adapted SafeStack, which provides effective protection against stack smashing and ROP attacks by isolating potentially exploitable stack variables from return addresses. While the security guarantees of the first two are deterministic, or by design, that of the last one is probabilistic and we evaluate its coverage.

1) Verifier: Each restricted operation is granted privileged execution, and in its original context this is desired and necessary. However, if the restricted operation is executed as part of a code reuse attack, the elevated privilege could undermine the security of the embedded system. To gain insight into the risk posed by the privilege overlays, we measure for each of the three IoT applications, how many overlays occur, how many instructions are executed in each, and how many have externally defined registers (external to the privilege overlay) that are used for addressing within the overlay. We wrote a verifier, which parses the assembly code of the application and identifies all privilege overlays. The results for the 20 variants of the IoT applications are shown in Table IV. It shows that the number of privilege overlays is small and that on average 5 to 7 instructions are executed within each. This results in a small attack surface and is a sharp reduction relative to the current state-of-practice in which the entire execution is in privileged mode.

2) Diversification: To further mitigate code reuse attacks and data corruption attack, EPOXY uses diversification for function locations in the code, data, and registers. This also provides protection against Data-oriented programming using

TABLE IV
RESULTS OF OUR VERIFIER SHOWING THE NUMBER OF PRIVILEGE OVERLAYS (PO), AVERAGE NUMBER OF INSTRUCTIONS IN AN OVERLAY (AVE), MAXIMUM NUMBER OF INSTRUCTIONS IN AN OVERLAY (MAX), AND THE NUMBER OF PRIVILEGE OVERLAYS THAT USE EXTERNALLY DEFINED REGISTERS FOR ADDRESSING (EXT).

App	PO	Ave	Max	Ext
PinLock	40	7.0	53	15
FatFs-uSD	31	5.0	20	0
TCP-Echo	25	5.2	20	0

global variables. Ultimately the amount of diversity available is constrained by the amount of memory. Our diversification strategies distribute any unused memory within the data, bss, and text regions. Let S denote the amount of slack memory and R denote the size of the region (any one of the three above, depending on which kind of diversification we are analyzing). For the text region S is the amount of unused Flash, and for the data and bss regions S is the amount of unused RAM. Then the total amount of memory available for diversifying any particular region is $R+S$—say for the global data region, the variable can be placed anywhere within R and the slack memory S can be split up and any piece "slid" anywhere within the data region. Since each is randomized by adding variables or jump instructions with a size of 4 bytes the total number of locations for a pointer is $(R+S)/4$.

Let us consider PinLock, our smallest example. It uses 2,376 bytes of RAM and would require a part with 4,096 bytes of RAM, leaving 1,720 bytes of slack. PinLock's data section is 1,160 bytes, thus a four byte pointer can have 720 locations or over 9 bits of entropy. This exceeds Linux's kernel level ASLR (9 bits, [29] Section IV), and unlike Linux's ASLR, disclosure of one variable does not reveal the location of all others. The text region is 11,788 bytes which means at least 16KB of Flash would be used. Since all Flash can be used except the region used for storing initial values for the data region (maximum of 1,556 bytes in PinLock), the text section can be diversified across 15,224 bytes. This enables approximately 3,800 locations for a function to be placed, which translates to entropy of just under 12 bits. Entropy is ultimately constrained due to the small size of memory but, similar to kernel ASLR, an attacker cannot repeatedly guess as the first wrong guess will raise a fault and stop the system.

3) ROP analysis: To understand how diversity impacts code reuse attacks we used the ROPgadget [52] ROP compiler. This tool disassembles a binary and identifies all the available ROP gadgets. A ROP gadget is a small piece of code ending in a return statement. It provides the building block for an attacker to perform ROP attacks. ROP attacks are a form of control hijack attacks which utilize only the code on the system, thus bypassing code integrity mechanisms. By chaining multiple gadgets together, arbitrary (malicious) execution can be performed. By measuring surviving gadgets across different variants we gain an understanding of how difficult it is for an attacker to build a ROP attack for a large set of binaries.

For each of the three applications, we identify gadgets individually in each of 1,000 variants. Each variant had all protections applied. To obtain the gadgets, ROPgadget parsed each file and reported all gadgets it found including duplicates. ROPgadget considers a duplicate to be the same instructions but at a different location, by including these we ensure that gadgets have the best chance of surviving across variants. The number of gadgets located at the *same location* with the *same instructions* were then counted across the 1,000 variants. To define the metric "number of gadgets surviving across x variants" consider a gadget that is found at the same location and with the identical instructions across *all* x variants. Count up all such gadgets and that defines the metric. This is a well-used metric because the adversary can then rely on the gadget to craft the control-flow hijacking attack across all the x variants. Clearly, as x goes up, this metric is expected to decrease. Table V shows the number of gadgets that survived across a given number of variants. To interpret this, consider that for the column "2", this number is the count of gadgets which survived across 2 or more variants of the program. The last remaining gadget survived across 48 variants of PinLock, only 32 variants of FatFS-uSD, and 107 variants of TCP-Echo. If a ROP attack only needs *the* single gadget which survives across the maximum number of variants—an already unlikely event—it would work on just over 10% of all variants. This shows that our code diversification technique can successfully break the attacker's ability to use the same ROP attack against a large set of binaries.

TABLE V
NUMBER OF ROP GADGETS FOR 1,000 VARIANTS THE IOT APPLICATIONS. LAST INDICATES THE LARGEST NUMBER OF VARIANTS FOR WHICH ONE GADGET SURVIVES.

		Num. Surviving				
App	Total	2	5	25	50	Last
PinLock	294K	14K	8K	313	0	48
FatFs-uSD	1,009K	39K	9K	39	0	32
TCP-Echo	676K	22K	9K	985	700	107

D. Comparison to FreeRTOS

Porting an application to FreeRTOS-MPU could provide some of the protections EPOXY provides. Compared to EPOXY, FreeRTOS-MPU does not provide $W \oplus X$ or code reuse defenses. FreeRTOS-MPU provides privilege separation between user tasks and kernel task.

User tasks running in unprivileged mode can access their stack and three user definable regions if it wishes to share some data with another user mode task. A kernel task runs in privileged mode and can access the entire memory map. A user task that needs to perform a restricted operation can be started in privileged mode but then the entire execution of the user task will be in privileged mode. If the privilege level is dropped, then it cannot be elevated again for the entire duration of the user task, likely a security feature in FreeRTOS-MPU.

We compare our technique to using FreeRTOS-MPU by porting PinLock to FreeRTOS-MPU. The vendor, STMicroelectronics, provided equivalent applications for FatFS-uSD and TCP-Echo that use FreeRTOS; we added MPU support to these application. This required: 1) Changing linker and startup code of the application to be compatible with the FreeRTOS-MPU memory map. 2) Changing the existing source code to use FreeRTOS-MPU specific APIs. 3) If any part of a task required a privileged operation, then the entire task must run with full privileges (*e.g.*, task initializing TCP stack).

Table VI shows the code size, RAM size, number of instructions executed and the number of privileged instructions for each application using EPOXY and FreeRTOS-MPU. The number of instructions executed (Exe) is the number of instructions executed for the whole application to completion.

TABLE VI
COMPARISON OF RESOURCE UTILIZATION AND SECURITY PROPERTIES OF FREERTOS-MPU(FREERTOS) VS. EPOXY SHOWING MEMORY USAGE, TOTAL NUMBER OF INSTRUCTIONS EXECUTED (EXE), AND THE NUMBER OF INSTRUCTIONS THAT ARE PRIVILEGED (PI).

App	Tool	Code	RAM	Exe	PI
PinLock	EPOXY	16KB	2KB	823K	1.4K
	FreeRTOS	44KB	30KB	823K	813K
FatFs-uSD	EPOXY	27KB	12K	33.3M	3.9K
	FreeRTOS	58KB	14KB	34.1M	33.0M
TCP-Echo	EPOXY	43KB	35KB	310.0M	1.5K
	FreeRTOS	74KB	51KB	321.8M	307.0M

Privileged instructions (PI) describe which of these instructions execute in privileged mode. Both are obtained using the Debug Watch and Trace unit provided by ARM [11]. The results for EPOXY are averaged over 100 runs across all 20 variants with 5 runs per variant, and FreeRTOS-MPU's are averaged over 100 runs. It is expected that the total number of instruction to be comparable as both are running the same applications. However, EPOXY uses an average of only 0.06% of privileged instructions FreeRTOS-MPU uses. This is because EPOXY uses a fine-grained approach to specify the privileged instructions, while FreeRTOS-MPU sets the whole task as privileged. A large value for PI is undesirable from a security standpoint because the instruction can be exploited to perform security-critical functions, such as, turning off the MPU thereby disabling all (MPU-based) protections.

VII. RELATED WORK

Our work uses our novel privilege overlays, to enable established security policies from the desktop world for bare-metal embedded systems. We also customize several of these protections to the unique constraints of bare-metal systems. Modern desktop operating systems such as Windows, Linux, and Mac OS X protect against code injection and control-flow hijack attacks through a variety of defenses, such as DEP [55], stack canaries [22], Address Space Layout Randomization [49], and multiple levels of execution privileges.

The research community has expended significant effort in developing defenses for control-flow hijacking and data corruption. These works include: Artificial Diversity [20, 36, 13, 14, 35, 32, 38, 41, 48, 25], Control-Flow Integrity (CFI) [9, 43, 58, 59, 46, 18], and Code Pointer Integrity (CPI) [40]. Artificial Diversity [20] outlines many techniques for creating functionally equivalent but different binaries and how they may impact the ability for attacks to scale across applications. A recent survey [41] performs an in-depth review of the 20+ years of work that has been done in this area. Artificial software diversity is generally grouped by how it is applied, by a compiler [36, 13, 14, 35, 32, 38, 45, 15] or by binary rewriting [48, 25]. With the exceptions of [32, 45, 15] these works target the applications supported by an OS, and assume virtual address space to create large entropy. Mclaughlin *et al.* [45] propose a firmware diversification technique for smart meters, using compiler rewriting. They give analytically results on how it would slow attack propagation through smart meters. They give no analysis with respect to execution time overhead or energy consumption. Giuffrida *et al.* [32] diversify the stack by adding variables to stack frames, creating a non-deterministic stack size which is not suitable for embedded systems. EPOXY applies compile-time diversification and utilizes techniques appropriate to their constraints. Braden *et al.* [15] focus on creating memory leakage resistant applications without hardware support. They use an approach based on SFI to prevent disclosure of code that has been randomized using fine-grained diversification techniques. Their approach assumes $W \oplus X$ and is compatible with MPUs. Our work provides a way to ensure enforcement of $W \oplus X$ automatically.

CFI uses control-flow information to ensure the targets of all indirect control-flow transfers end up at valid targets. CFI faces two challenges: precision and performance. While the performance overhead has been significantly reduced over time [46, 54], even the most precise CFI mechanism is ineffective if an attacker finds a code location that allows enough gadgets to be reached, e.g., an indirect function call that may call the function desired by the attacker [19, 28]. CFI with custom hardware additions has been implemented on embedded systems [24] with low overhead. Our techniques only require the commonly available MPU. CPI [40] enforces strict integrity of code pointers with low overhead but requires runtime support and virtual memory. However, separate memory regions and MMU-based isolation are not available on bare-metal embedded systems. We leverage SafeStack, an independent component of CPI that protects return addresses on the stack, and adapt it to embedded systems without virtual memory support.

Embedded systems security is an important research topic. Cui and Stolfo [23] use binary rewriting to inject runtime integrity checks and divert execution to these checks; diversifying code in the process. Their checks are limited to checking static memory via signatures and assumes DEP. Francillon *et al.* [31] use micro-controller architecture extensions to create a regular stack and a protected return stack. EPOXY also uses a dual stack, without additional hardware support. Firmware integrity attestation [30, 27, 44, 10] uses either a software or hardware trust anchor to provide validation that the firmware and or its execution matches a known standard. These techniques can be used to enforce our assumption that the firmware is not tampered with at installation. Some frameworks [16, 4, 7, 1] enable creation of isolated computational environments on embedded systems. mbedOS[4] and FreeRTOS [1] are both embedded operating systems which can utilize the MPU to isolate OS context from application context. TyTan [16] and mbed $\mu Visor$ [7] enable sandboxing between different tasks of a bare-metal system. These require that an application be developed using its respective API. ARM's TrustZone [12] provides hardware to divide execution between untrusted and trusted execution environments. The ARMv7-M architecture does not contain this feature.

VIII. Discussion

Real-time systems. The diversity techniques we employ introduce some non-determinism between variants. This may make it unsuitable for real-time systems with strict timing requirements. However, the variability is low (a few percent) making our techniques applicable to wide ranges of devices, particularly IoT devices, as they generally have *soft* real-time constraints. Investigation of the methods to further reduce variability is an area of future work. This involves intrusive changes to the compiler infrastructure to make its actions more deterministic in the face of diversification.

Protecting inputs and outputs. We demonstrated EPOXY's ability to protect the lock actuator on PinLock. Protecting the Ethernet and the SD interfaces is conceptually the same—a series of reads and writes to IO registers. However, the HAL for these interfaces makes use of long indirection chains, *i.e.*, passing the addresses of these registers as function parameters. Our current analysis does not detect these accesses, and the complexity of the HAL makes manual annotation a daunting task. Extending our analysis to be inter-procedural will allow us to handle these complex IO patterns.

Use with lightweight OSs. EPOXY can be extended to apply its protections to lightweight OSs, such as FreeRTOS. Our diversity techniques are directly usable as they do not change any calling conventions. Privilege Overlays require the use of a system call and care must be take to ensure one is reserved. Currently SVC FE is used, an arbitrary choice, which can be changed to a compile-time parameter. Thus, enabling the application of $W \oplus X$—assuming the OS does not use the MPU, which typically is the case. To apply SafeStack, the only remaining protection, EPOXY needs to know the number of threads created, and how to initialize each *unsafestack*. This may be obtained by making EPOXY aware of the OS thread create functionality, so it can be modified to setup both stacks. The OS's context switch would also need to be changed to save and restore separate *unsafestack* guards for each thread. With these changes EPOXY could apply its defenses to systems using a lightweight OS.

IX. Conclusion

Bare-metal systems typically operate without even basic modern security protections like DEP and control-flow hijack protections. This is caused by the dichotomy inherent in bare-metal system development: all memory is executable and accessible to simplify system development, but security principles dictate restricting some of their use at runtime. We propose EPOXY, that uses a novel technique called privilege overlaying to solve this dichotomy. It applies protections against code injection, control-flow hijack, and data corruption attacks in a system-specific way. A performance evaluation of our prototype implementation shows that not only are these defenses effective, but that they result in negligible execution and power overheads. The open-source version of EPOXY is available at https://github.com/HexHive/EPOXY.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also thank Brandon Eames for his informative feedback. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1464155 and CNS-1548114. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work is also funded by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] FreeRTOS-MPU. http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html
[2] FreeRToS Support Forum. ARM_CM3_MPU does not seem to build in FreeRTOS 9.0.0. https://sourceforge.net/p/freertos/discussion/382005/thread/3743f72c/
[3] FreeRToS Support Forum. Stack overflow detection on Cortex-m3 with MPU . https://sourceforge.net/p/freertos/discussion/382005/thread/18f8a0ce/#deab
[4] mbed OS. https://www.mbed.com/en/development/mbed-os/
[5] STM32479I-EVAL. http://www.st.com/resource/en/user_manual/dm00219352.pdf
[6] STM32F4-Discovery. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00037955.pdf
[7] The mbed OS uVisor. https://www.mbed.com/en/technologies/security/uvisor/
[8] FreeRToS Support Forum. Mistype in port.c for GCC/ARM_CM3_MPU , Jan 2016. https://sourceforge.net/p/freertos/discussion/382005/thread/6a4f7df2/
[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, Control-flow integrity, In *ACM Conf. on Computer and Communication Security*. ACM, 2005, pp. 340–353.
[10] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, C-FLAT: control-flow attestation for embedded systems software, In *Symp. on Information, Computer and Communications Security*, 2016.
[11] ARM, *ARMv7-M Architecture Reference Manual*, "E.b" ed., 2014.
[12] ARM, Trustzone, 2015. http://www.arm.com/products/processors/technologies/trustzone/
[13] S. Bhatkar, D. DuVarney, and R. Sekar, Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. *USENIX Security Symp.*, 2003.
[14] S. Bhatkar, D. DuVarney, and R. Sekar, Efficient Techniques for Comprehensive Protection from Memory Error Exploits, *USENIX Security Symp.*, 2005.
[15] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, Leakage-resilient layout randomization for mobile devices, In *Network and Distributed Systems Security Symp. (NDSS)*, 2016.
[16] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, Tytan: Tiny trust anchor for tiny devices, In *Design Automation Conf.* ACM/IEEE, 2015, pp. 1–6.
[17] bunnie and Xobs, The exploration and explotation of a sd memory card, In *Chaos Computing Congress*, 2013.
[18] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, Control-Flow Integrity: Precision, Security,

and Performance, *ACM Computing Surveys*, vol. 50, no. 1, 2018, preprint: https://arxiv.org/abs/1602.04056.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, In *SEC: USENIX Security Symposium*, 2015.

[20] F. B. Cohen, Operating system protection through program evolution, *Computers and Security*, vol. 12, no. 6, pp. 565–584, oct 1993.

[21] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, A large-scale analysis of the security of embedded firmwares, In *USENIX Security Symp.*, 2014, pp. 95–110.

[22] C. Cowan, C. Pu, D. Maier, and J. Walpole, StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *USENIX Security Symp.*, 1998.

[23] A. Cui and S. J. S. Stolfo, Defending Embedded Systems with Software Symbiotes, In *Intl. Conf. on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 358–377.

[24] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, Hafix: Hardware-assisted flow integrity extension, In *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15, 2015, pp. 74:1–74:6.

[25] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, Gadge Me If You Can, In *Symp. on Information, Computer and Communications Security*. ACM Press, 2013, p. 299.

[26] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain, Can you still trust your network card, *CanSecWest*, pp. 24–26, 2010.

[27] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed System Security Symp.*, vol. 12, 2012, pp. 1–15.

[28] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, Control jujutsu: On the weaknesses of fine-grained control flow integrity, In *CCS'15: Conference on Computer and Communications Security*, 2015.

[29] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, Jump over aslr: Attacking branch predictors to bypass aslr, In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[30] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, A minimalist approach to remote attestation, In *Euro. Design, Automation, and Test*. EDAA, 2014, p. 244.

[31] A. Francillon, D. Perito, and C. Castelluccia, Defending embedded systems against control flow attacks, In *ACM Conf. on Computer and Communication Security*, 2009, pp. 19–26.

[32] C. Giuffrida, A. Kuijsten, and A. Tanenbaum, Enhanced operating system security through efficient and fine-grained address space randomization. *USENIX Security Symp.*, 2012.

[33] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, The mälardalen wcet benchmarks: Past, present and future, In *Open Access Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, In *Intl. Work. on Workload Characterization*. IEEE, 2001, pp. 3–14.

[35] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, Profile-guided automated software diversity, In *Intl Symp. on Code Generation and Optimization*. IEEE, 2013, pp. 1–11.

[36] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, Librando: Transparent code randomization for just-in-time compilers, In *ACM Conf. on Computer and Communication Security*, 2013.

[37] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, Data-oriented programming: On the expressiveness of non-control data attacks, In *IEEE Symp. on Security and Privacy*. IEEE, 2016, pp. 969–986.

[38] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, Compiler-generated software diversity, In *Moving Target Defense*. Springer, 2011, pp. 77–98.

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

[40] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, Code Pointer Integrity, *USENIX Symp. on Operating Systems Design and Implementation*, 2014.

[41] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, SoK: Automated Software Diversity, *IEEE Symp. on Security and Privacy*, pp. 276–291, 2014.

[42] C. Lattner and V. Adve, Llvm: A compilation framework for lifelong program analysis and transformation, In *Intl. Symp. Code Generation and Optimization*. IEEE, 2004, pp. 75–86.

[43] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, Comprehensive and efficient protection of kernel control data, *IEEE Trans. on Information Forensics and Security*, vol. 6, no. 4, pp. 1404–1417, 2011.

[44] Y. Li, J. M. McCune, and A. Perrig, Viper: Verifying the integrity of peripherals' firmware, In *ACM Conf. on Computer and Communications Security*, 2011, pp. 3–16.

[45] S. E. McLaughlin, D. Podkuiko, A. Delozier, S. Miadzvezhanka, and P. McDaniel, Embedded firmware diversity for smart electric meters. In *USENIX Work. on Hot Topics in Security*, 2010.

[46] B. Niu and G. Tan, Modular control-flow integrity, *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[47] J. Pallister, S. J. Hollis, and J. Bennett, BEEBS: open benchmarks for energy measurements on embedded platforms, *CoRR*, vol. abs/1308.5174, 2013.

[48] V. Pappas, M. Polychronakis, and A. D. Keromytis, Smashing the gadgets: Hindering return-oriented programming using in-place code randomization, *IEEE Symp. on Security and Privacy*, pp. 601–615, 2012.

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

[50] G. Ramalingam, The undecidability of aliasing, *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, Sep. 1994.

[51] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, Security and privacy challenges in industrial internet of things, In *Design Automation Conf.* ACM/IEEE, 2015, p. 54.

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

[53] L. Szekeres, M. Payer, and D. Song, SoK: Eternal War in Memory, In *IEEE Symp. on Security and Privacy*. IEEE, may 2013, pp. 48–62.

[54] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, Enforcing forward-edge control-flow integrity in gcc & llvm, In *USENIX Security Symp.*, 2014.

[55] A. van de Ven and I. Molnar, Exec Shield, 2004. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

[56] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, Efficient software-based fault isolation, In *SOSP'03: Symposium on Operating Systems Principles*, 1993.

[57] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, Implementation and implications of a stealth hard-drive backdoor, In *Annual Computer Security Applications Conf.*, 2013, pp. 279–288.

[58] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, Practical control flow integrity and randomization for binary executables, In *IEEE Symp. on Security and Privacy*. IEEE, 2013, pp. 559–573.

[59] M. Zhang and R. Sekar, Control flow integrity for cots binaries, In *USENIX Security Symp.*, 2013, pp. 337–352.

[60] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, Dspstone: A dsp-oriented benchmarking methodology, In *Intl. Conf. on Signal Processing Applications and Technology*, 1994, pp. 715–720.