Challenges in Designing Exploit Mitigations for Deeply Embedded Systems

Ali Abbasi∗, Jos Wetzels†, Thorsten Holz∗ and Sandro Etalle†
∗ Ruhr-University Bochum, Germany
Email: ali.abbasi-i4q@rub.de, thorsten.holz@rub.de
† Eindhoven University of Technology, The Netherlands
Email: a.l.g.m.wetzels@student.tue.nl, s.etalle@tue.nl

Abstract—Memory corruption vulnerabilities have been around for decades and rank among the most prevalent vulnerabilities in embedded systems. Yet this constrained environment poses unique design and implementation challenges that significantly complicate the adoption of common hardening techniques. Combined with the irregular and involved nature of embedded patch management, this results in prolonged vulnerability exposure windows and vulnerabilities that are relatively easy to exploit. Considering the sensitive and critical nature of many embedded systems, this situation merits significant improvement.

In this work, we present the first quantitative study of exploit mitigation adoption in 42 embedded operating systems, showing the embedded world to significantly lag behind the general-purpose world. To improve the security of deeply embedded systems, we subsequently present µArmor, an approach to address some of the key gaps identified in our quantitative analysis. µArmor raises the bar for exploitation of embedded memory corruption vulnerabilities, while being adoptable on the short term without incurring prohibitive extra performance or storage costs.

I. INTRODUCTION

Embedded systems are everywhere, from simple networking equipment to satellite systems. With the rise of cyber-physical systems and the Internet of Things (IoT), these systems are set to proliferate throughout all aspects of everyday life. Due to their ubiquitous and often sensitive and critical nature, embedded systems pose many security and privacy concerns. Unfortunately, proper attention to security in the embedded world tends to be scarce in practice. This tendency becomes clear from various studies [1], [2] revealing security flaws in a wide variety of embedded systems. These flaws are far from hypothetical. One example is the high-profile attack on embedded systems known for having been used to construct the IoT-powered botnet Mirai [3]. Yet embedded systems security is seen as lagging behind what we have come to expect of our general purpose (e.g., desktop and server) systems [4].

Embedded binary security, in particular, is an area where exploitation of vulnerabilities is significantly easier than on general-purpose systems. This is exemplified by a recent incident where a previously unknown group calling themselves the Shadow Brokers released a cache of exploits which they claimed belonged to the supposedly state-sponsored Equation Group [5] threat actor. Among this cache was a set of exploits for high-end firewall equipment, none of which had to bypass any exploit mitigations. Despite these threats, and the general perception of embedded systems binary security as lagging, we do not yet have a clear understanding of the existing gap in security technologies available for embedded systems compared to general-purpose computers.

In this paper, we study this problem and focus on three major bare-minimum and well-known exploit mitigation techniques. More specifically, we focus on Executable Space Protection (ESP, also known as NX, DEP, or W⊕X policy), Address Space Layout Randomization (ASLR) and stack canaries to identify the gap in security technologies specifically in the memory corruption and exploitation domain. These exploit mitigation methods are almost universally available on general-purpose computers. We investigate whether such mitigations are available in 42 major embedded Operating Systems (OS). We found that half of them provide such mitigations and the majority of them belong to the group of so-called high-end embedded OSes. However, when it comes to lower-end embedded OSes that are mostly being used in so-called deeply embedded systems, exploit mitigations are almost absent. The deeply embedded systems as described by Koopman et al. [6] are a subset of embedded systems which usually rely on 8-, 16- or (at the higher end of the spectrum) 32-bit micro-controllers. These systems tend to come in the form of a micro-controller unit (MCU) or more extensive System-on-Chip (SoC) devices, embedding both the core as well as memory and peripheral devices into a single chip.

We investigated 78 common [7] embedded microprocessors and microcontrollers to understand whether this universal lack of exploit mitigations is caused by lack of hardware feature support (i.e., Memory Management Unit (MMU), Memory Protection Unit (MPU), or Hardware-supported ESP).

While the lack of exploit mitigations for bare-metal embedded systems (i.e., devices without an OS) is addressed in recent research [8], no research suggests a solution for deeply embedded systems that are running multi-stack, multi-threaded and real-time capable operating systems. This lack of solutions comes with risks that are expected to increase. According to VDC Research [9], the IoT has caused developers for deeply embedded systems to move away from bare-metal embedded systems towards deeply embedded systems running an OS.

Based on our quantitative analysis on the lack of exploit...
mitigation support on deeply embedded OSes and considering the recent research by Celements et al. [3] which addresses bare-metal deeply-embedded systems, we introduce µArmor, a set of LLVM passes that harden deeply embedded systems that are running an OS. Our goal is to bring exploit mitigation baselines from general-purpose computers to the most constrained end of the embedded spectrum. More specifically, µArmor adds several exploit mitigation strategies to such systems. We have built a prototype implementation of this concept and evaluate the overhead imposed by µArmor in terms of code size, data size, memory usage, and runtime overhead. Our empirical results indicate that the induced overhead is very low and suitable even for deeply embedded systems.

In summary, we make the following contributions:

- We perform a comprehensive, quantitative study of exploit mitigation adoption in 42 embedded operating systems. This analysis clearly shows that embedded systems severely lag behind general purpose ones.
- We perform a systematic identification of the challenges faced by embedded exploit mitigation adoption efforts. Based on this analysis, we identify two major open problems and subsequently introduce a solution to address them.
- We propose, implement and evaluate µArmor, an exploit mitigation baseline design for deeply embedded systems running a multi-stack, multi-threaded, real-time capable operating system. Additionally, as part of µArmor, we present µSSP, a stack canary scheme with a modular violation policy handler allowing for the preservation of the system’s availability.

II. Mitigations Baseline

The term embedded systems covers a large number of different devices that are dedicated to a specific purpose. In this work, we focus on so-called deeply embedded systems [6], a subset of embedded systems which usually rely on 8-, 16- or (at the higher end of the spectrum) 32-bit micro-controllers. These systems tend to come in the form of a micro-controller unit (MCU) or more extensive System-on-Chip (SoC) devices, embedding both the core as well as memory and peripheral devices into a single chip. Such a high level of integration allows for low production cost and simplifies production, but at the same time constrains capabilities with regards to memory size, speed and power consumption. Deeply embedded systems often lack user interfaces or have uncommon ones and tend to run extremely minimal OSes (often with real-time capabilities) or no OS at all (bare-metal).

We are interested in defining the absolute minimum set of mitigations which should be reasonably expected to be present in all modern embedded systems. Because of the sheer diversity of embedded systems, we do not select our baseline by strict criteria. Instead, we require them to be adaptable across the embedded spectrum and not rely on any specialized hardware feature not commonly present in Commercial off-the-shelf (COTS) embedded hardware. The minimum embedded exploit mitigation baseline we selected comprised of ESP, ASLR, and stack canaries. These mitigations were selected because they are complementary and have been integrated into virtually all modern general-purpose OSes and development toolchains, including those widely used in the embedded world. As such they are well understood and can reasonably be considered to be the absolute minimum in modern exploit mitigations.

III. Quantitative Analysis of Exploit Mitigation Methods in Embedded Systems

In this section, we present a quantitative evaluation of exploit mitigation adoption (as per our baseline outlined in Section II) and dependency support among popular embedded operating systems and hardware. The results of our quantitative evaluation reflect the current embedded state-of-the-art, allowing us to identify clear gap-areas and new opportunities to improve the security of such systems.

A. Embedded OS Mitigation and Dependency Support

We evaluated 42 popular embedded operating systems to present an overview of the current state of embedded OS mitigation adoption. Our selection aims to be a representative sample of embedded operating systems and includes those listed by the UBM Embedded Markets Study [7], and various studies into embedded operating systems [10], [11], [12] as well as some of the most popular mobile operating systems [13].

We evaluated these 42 embedded OS for exploit mitigation and dependency support through a combination of vendor surveys, documentation consultation, and experimental validation. An overview of the results (including the list of OSes) is shown in Tables II and III while aggregated results are shown in Tables III and IV.

Note that we consider a mitigation or feature supported iff it is supported by the OS for at least some (but not necessarily all) platforms. Since this is a quantitative assessment, it neither evaluates the quality of the implementation nor whether the feature is enabled by default and as such the assessment is an optimistic one. We mark an OS as providing a CSPRNG (Cryptographically Secure Pseudo-Random Number Generator) iff provided PRNG (Pseudo-Random Number Generator) functionality is advertised as such or can be reasonably assumed to provide secure random number generation functionality.

B. Embedded Hardware Feature Support

1) Von Neumann vs Harvard: Before we can discuss the hardware features support for exploit mitigations in embedded systems, we first need to consider that there are essentially two main processor architectural styles: Harvard and Von Neumann. The Harvard CPU architecture has separated instruction and data busses and thus allows operations to run simultaneously, while physically separating signals and storage for code and data memory. In contrast, the Von Neumann CPU architecture has only one bus which is used for both data transfers and instruction fetches, thus any value in memory can be executed or interpreted as data, respectively.
Due to the sensitive nature of RedactedOS, we have received it on the condition of anonymity for the vendor. RedactedOS is a real-time OS which is primarily being used for aerospace applications.

2) Hardware Feature Support: The embedded world features a wide range of different processor architectures and core families with different capabilities. To establish an overview of common embedded hardware capabilities, we make a selection of several core families and map out their architectural style and MPU, MMU, and hardware ESP support capabilities.

We evaluated 78 different core families for hardware dependency support. 57 of core family SoCs were based on Von Neumann architecture and 21 of them were based on Harvard architecture. An overview of the supported feature detailed results reported in Tables V and VI. Our selection of core families aims to be a representative sample of core families belonging to major architectures and vendors in the embedded space across industry verticals and includes, among others, the most popular core families listed by recent UBM Embedded Markets Studies [7] and EDN reader surveys [14].

TABLE I: Detailed Embedded OS exploit mitigation adoption. Red for Mobile embedded OSes, white for regular embedded OSes and gray for deeply embedded OSes.

| OS                          | ESP | ASLR | Canaries | OS                          | ESP | ASLR | Canaries |
|-----------------------------|-----|------|----------|-----------------------------|-----|------|----------|
| BlackBerry OS               | ✓   | ✓    |          | Android*                    | ✓   | ✓    |          |
| Alan*                       | ✓   | ✓    |          | Win 10 Mobile*              | ✓   | ✓    |          |
| Sailfish OS*                | ✓   | ✓    |          | Tiara*                      | ✓   | ✓    |          |
| Ubuntu Core                | ✓   | ✓    |          | Brillo*                     | ✓   | ✓    |          |
| Yocto Linux*               | ✓   | ✓    |          | Windows Embedded*           | ✓   | ✓    |          |
| OpenWRT*                    | ✓   | ✓    |          | NanoOS*                     | ✓   | ✓    |          |
| µChips                     | ✓   | ✓    |          | FreeRTOS                    | ✓   | ✓    |          |
| NoxBSD®                    | ✓   | ✓    |          | Interos Zero RTX*           | ✓   | ✓    |          |
| SymbOS®                    | ✓   | ✓    |          | TermOS®                     | ✓   | ✓    |          |
| QNX                        | ✓   | ✓    |          | Cryptos                     | ✓   | ✓    |          |
| Integrity                  | ✓   | ✓    |          | RedactedOS 2                 | ✓   | ✓    |          |

Embedded OS based on Windows, Linux or BSD.

TABLE II: Detailed Embedded OS exploit mitigation dependency support for Memory Protection (MPROT), Virtual Memory (VMEM) and Random Number Generator (RNG). Red for Mobile embedded OSes, white for regular embedded OSes and gray for deeply embedded OSes.

* Embedded OS based on Windows, Linux or BSD.

TABLE III: Overview of Embedded OS Exploit Mitigation Support.

| OS vs. Mitigation | ESP | ASLR | Stack Canaries |
|-------------------|-----|------|----------------|
| All evaluated OSes | 2042 | 1342 | 1742 |
| Non-Mobile        | 16/16 | 8/36 | 12/36 |
| Non-Linux/Windows/BSD | 7/22 | 1/27 | 5/27 |
| Deeply Embedded   | 3/20 | 0/20 | 1/20 |

TABLE IV: Overview of Embedded OS Exploit Mitigation Dependency Support.

| Core Family | Arch. | MPU | MMU | ESP |
|-------------|-------|-----|-----|-----|
| ARM         | N     | x   | x   | x   |
| ARM1        | N     | x   | x   | x   |
| ARM2        | N     | x   | x   | x   |
| ARM3        | N     | x   | x   | x   |
| ARM6        | N     | x   | x   | x   |
| ARM7        | N     | ~   | ~   | x   |
| ARM7EJ      | N     | x   | x   | x   |
| ARM8        | N     | x   | x   | x   |
| ARM9T       | N     | ~   | ~   | x   |
| ARM10E      | N     | x   | ~   | ~   |
| ARM11       | N     | ~   | ~   | ~   |
| ARM Cortex-A | N | ✓ | ✓ | ✓ |
| ARM Cortex-R | N | ✓ | ✓ | ✓ |
| ARM Cortex-M | N | ~ | ~ | ~ |

TABLE V: Core Family dependency support in Harvard (H) and Von Neumann (N) architectures. We consider a feature supported if it is supported by all members of a given core family and absent if it is not supported by any of them. Any variation with regards to dependency support is denoted with ~ and omitted from aggregated results.

The core families in our selection belong to the following embedded architectures: ARM, MIPS32, PIC, PPC, x86,
SuperH, AVR, AVR32, Intel 8051, Motorola 68000, TriCore, MSP430, C166, Blackfin, ARC, Renesas Electronics RL78 and RX.

C. Quantitative Analysis Results

Among the embedded OSes surveyed, we can distinguish two major clusters in terms of capabilities and purposes:

- **High-End**: These OSes are aimed at the higher end of the embedded spectrum and offer virtual memory capabilities as well as often being POSIX-compliant. This includes mobile OSes (e.g., Android and iOS), lightweight versions of OSes common in the general-purpose world (such as Linux, Windows or BSD) as well as OSes like QNX or VxWorks.

- **Low-End**: These OSes are aimed at deeply embedded systems, often have real-time capabilities and do not offer virtual memory support. As such, there is usually no separation between user- and kernelspace and instead of isolated processes, there tends to be just a kernel running a limited set of tasks. Examples are Real-Time Operating Systems (RTOSes) such as ThreadX, RTEMS, Micrium μCOS and TinyOS.

From the results in Tables III and IV we can observe that all mobile OSes have support for every exploit mitigation in our baseline and so do most Linux, BSD, and Windows-based OSes. Outside of those, however, almost all other OSes (apart from QNX) lack support for any mitigations whatsoever. We can also see that while memory protection support is almost universally present, virtual memory and OS CSPRNG support is almost universally lacking in the low-end OSes aimed at deeply embedded systems. From these observations we can conclude that exploit mitigation adoption (and underlying dependency support) is generally present only on the high-end embedded OSes which derive from Linux, BSD or Windows.

When it comes to the hardware core families surveyed, we can see that less than half of the (Von Neumann) core families in our selection have MPU support. A small minority of core families has MPU support (MPU was supported in 6 out of 51 core families in our selection without necessary hardware support required for virtual memory). This lack of MPU and MMU support makes sense for the more constrained deeply embedded systems (such as Linux, Windows or BSD) as well as OSes like QNX or VxWorks.

### TABLE VII: Overview of hardware dependency support in Von Neumann core families.

| Core Family | Arch. | MPU | MMU | ESP |
|-------------|-------|-----|-----|-----|
| x86         |       |     |     | ✓   |
| Intel Atom Z34XX | N × ✓ ✓ |
| Intel Quark X10XX | N × ✓ ✓ |
| Intel Quark μC² | N × ✓ ✓ |
| SuperH      |       |     |     |     |
| SH-1        | N × x x |
| SH-2        | N × x x |
| SH-3        | N × ✓ x |
| SH-4        | N × ✓ x |
| AVR         |       |     |     |     |
| AVR32       | N ✓ x x |
| AVR32B      | N ✓ x x |
| 8051        | Intel MCS-51 | H × x x |
| Infineon XCS8X-I | H x x x |
| Infineon XCS8X-A | H x x x |
| ARM         |       |     |     |     |
| TriCore     | Infineon T1010 | H x x x |
| Infineon AUDO Future | H x x x |
| C166        | Infineon XFl66 | N ✓ x x |
| Infineon XC2200 | N ✓ x x |
| MSP430      |       |     |     |     |
| MSP4301xx   | N × x x |
| MSP4302xx   | N × x x |
| MSP4303xx   | N × x x |
| MSP4304xx   | N × x x |
| MSP4305xx   | N × x x |
| MSP4306xx   | N × x x |
| MSP430FRxx  | N ✓ x x |
| Blackfin    | Analog Blackfin² | N ✓ x x |
| ARC         | Synopsys ARC 1M | H x x x |
| Synopsys ARC 600 | H ~ x x |
| Synopsys ARC 700 | H x x x |
| RL78        | Renesas RL78/8Fx | H x x x |
| Renesas RL78/4Lx | H x x x |
| RX          | Renesas RX2600 | H x x x |
| Renesas RX600 | H ~ x x |

1 Intel Quark Microcontrollers (D1000/C1000/D2000)

2 Although documentation mentions an MMU, it does not support address translation (and thus does not allow for virtual memory) which is why we consider it an MPU for our purposes.
when it comes to low-end OS capabilities, the lack of virtual memory and cryptographically secure pseudorandom number generator (CSPRNG) support present obstacles to ASLR and stack canary adoption.

IV. CHALLENGES FOR EMBEDDED SYSTEMS

To explain the gaps in embedded exploit mitigation adoption and implementation discussed in Section III, we now discuss the challenges faced by embedded systems developers for integrating mitigation within their software. Based on this discussion, we will identify a series of open problems in the field of embedded exploit mitigations and outline the design criteria for exploit mitigations and OS CSPRNGs for deeply embedded systems. Generally we can divide the reasoning behind lack of exploit mitigation within embedded system to the following groups:

1) Development Practices & Cost Sensitivity
2) Resource Constraints
3) Safety, Reliability & Real-Time Requirements
4) Hardware & OS Limitations

We discuss these open problems to understand why a mitigation is not available even though the required hardware features exist or why hardware features are missing in practice.

A. Development Practices & Cost Sensitivity

Embedded systems development practices and design cultures are different from those in desktop or web application development. Compared to the general-purpose world, the embedded world is heavily fragmented among many different vendors and suppliers and technologies themselves are fragmented into competing standards without clear market leaders and individual solutions for specific problems. For example, a typical embedded product is put together as the result of a hardware vendor selling a chip, deployed with an operating system and some drivers, to an embedded systems manufacturer (the Original Equipment Manufacturer (OEM)) who integrates it into the embedded system in question (adding some hardware peripherals, writing some software) and often resells it to a brand-name company who adds a user- or machine-to-machine interface and put it on the consumer market. This fragmentation leads to the following issues:

- **Lowest Common Denominator** Vendors at the top of the chain such as chip or embedded operating system vendors often cater to very diverse customers and as such are bound by the demands (in terms of capabilities, overhead and cost increases) of their most constrained customers.

- **Fragmented Security Requirements** No single entity oversees the entire software development life-cycle and as such, there is no coherent, single set of security requirements.

- **Patching & Maintenance Issues** In many cases no single entity has the ability to patch or upgrade every piece of software on a given embedded device once it’s shipped.

- **Incentive Issues** While there might be a strong case for certain security measures when considering the end product as a whole, there is usually little incentive on part of individual vendors and manufacturers who are just a single link in a much bigger chain. Embedded systems markets are often characterized by a heavy focus on time-to-market (earlier market introduction tends to mean deeper market penetration and hence higher potential revenue) and novel features: since embedded systems are designed for a specific purpose rather than general-purpose computing, vendors often differentiate themselves on the basis of price and specific features rather than generic capabilities unrelated to the specific utility of the end product. As a result, there is little incentive for integrating security measures if these are not already present by default.

- **Cost Sensitivity** Embedded systems are often very cost sensitive, they tend to be produced in large quantities and as such even small cost increases per unit rapidly amount to large overall production cost increases. In addition, for the cheaper products a cost increase on part of a single component soon amounts to a higher percentage of total system cost, making it harder to justify such a cost increase, especially with something that is often so hard to quantify as improved security. Finally, any cost savings might aid in gaining a market advantage for price sensitive products. As such there usually is a preference for cheaper, simpler hardware such as chips with few features and limited room for overhead. This matters from a security perspective because it makes many hardware-based security measures infeasible (because of the associated per-unit cost increases) and means designers and implementers of embedded security measures have to deal with limited hardware capabilities and resource constraints.

B. Resource Constraints

Generally, embedded systems face significant resource constraints since they are designed with a specialized, dedicated purpose in mind rather than aiming to provide general-purpose capabilities. As such, all resources considered superfluous to this task are eliminated to reduce production cost which results in limitations on code and data memory, processing power and hardware capabilities. Embedded software, in turn, is designed to be efficient and have a minimal footprint in order to meet these constraints given the limited room for overhead. These constraints are including code storage size, memory size, processing power, and power consumption. An example of impact of mentioned constrains on exploit mitigation is power consumption. This constraint immediately conflicts with security measures that introduce power consumption overhead (e.g., due to being computationally intensive or requiring “power hungry” hardware).

In the context of this work we concern ourselves with four major resource constraint areas:

1) **Code Storage Size**: This constraints limit code size overhead and the introduction of additional functionality. Many embedded systems are diskless and do not have permanent storage, storing code in flash memory of a
few KB or MB instead. Those systems that do have permanent storage use something like a few KB of EEPROM, usually to store configuration data only since the infrequent changing of code means it is more economical to arrange this via flashing a firmware update, or are far more limited than hard disk capacities of desktop or server systems (e.g., using SD cards of a few GB).

2) **Memory Size:** This constraints limit memory usage overhead and often rule out the possibility of memory-intensive computations. Embedded systems, particularly deeply embedded systems, often do not have external memory but rely only on a few KB or MB of on-chip internal (S)RAM. Those systems that do have external memory are often limited to anything from a few dozen MB up to one or two GB.

3) **Processing Power:** While there is a trend towards usage of more powerful 32-bit processors [15], [7], [16] running at clock speeds ranging from 100 MHz to around 1 GHz (the average in 2015 being 397 MHz according to [7]) and there are plenty of embedded segments where even more serious computing power is a must, many embedded systems continue to use simpler 8- or 16-bit processors with clock speeds ranging from 8 to 32 MHz. Such a lack of processing power inhibits deployment of computationally intensive security measures and certain cryptographic algorithms.

4) **Power Consumption:** Many embedded systems have serious power consumption constraints [21], [4] as a result of being battery operated, having to last months, years or indefinitely on a single battery while others might get recharged more frequently. As such, this constraint conflicts with security measures that introduce significant power consumption overhead (as mentioned above).

### C. Safety, Reliability & Real-Time Requirements

Embedded systems tend to have specific requirements relating to safety, reliability, and real-time computation [6]:

a) **Safety & Reliability:** Some embedded systems have stringent safety and reliability requirements which would require certification of any security measures upon their introduction and require them to be robustly reliable (e.g., maintain availability). This means, for example, that exploit mitigations for these embedded systems will have to avoid invocation of alert policies that violate safety and reliability (such as abruptly terminating critical software upon detection of attacks [22], [23]).

b) **Timeliness:** Many embedded systems are subject to varying degrees of hardness real-time requirements and use real-time operating systems (RTOS) to accommodate this. As such, security measures for those systems will need to respect those requirements [22]. However, such requirements might inherently conflict with certain exploit mitigation designs or their dependencies. Consider, for example, ASLR and its dependency on virtual memory. Traditionally, the use of virtual memory in real-time operating systems has been avoided due to timing analysis complications [24]. Virtual memory poses predictability problems regarding worst-case execution time (WCET) analysis largely because of two issues [24], [25]:

1) **Address Translation:** Mapping virtual to physical addresses is commonly done using a translation look-aside buffer (TLB): a memory cache that is part of the MMU and stores recent address translations. However, address translation timings are hard to predict, because a) not all mappings are cached in the TLB leading to cache misses requiring a subsequent page table lookup and b) the TLB is shared between different processes.

2) **Paging:** Since physical memory is shared between different processes and any physical page may be selected for replacement by the paging algorithm, predicting whether a virtual memory reference results in a page fault is hard. In addition, paging makes memory access timings dependent on TLB and cache contents increasing unpredictability. Finally, page faults may incur significant overhead rendering a system non-responsive for too long.

Various hardware/software-based proposals for real-time compatible virtual memory exist [24], [25], but to the best of our knowledge, none of these have been adopted by popular RTOSs due to significant performance penalties or hardware cost increases.

### D. Hardware & OS Limitations

As a result of the embedded cost sensitivity and resource constraints discussed above, embedded hardware and operating systems are often lacking the features upon which modern security measures depend. We will briefly discuss the implications of these limitations for the future embedded adoption of the exploit mitigations in our baseline as well as identify some related open problems.

1) **MPUs, MMUs & Hardware ESP:** As shown in Table VII under half of surveyed embedded core families have hardware ESP support. While 32-bit processors are clearly gaining increasing traction within the embedded world and are even displacing 8- and 16-bit processors [15], smaller 8-bit processors continue to dominate a significant portion of the embedded space. While many modern 32-bit processors tend to be Von Neumann and while many popular architectures in this category have hardware ESP support (e.g., ARMv6+, MIPS32r3+) there are others which do not. Even though for many systems based on those smaller 8- and 16-bit processors it’s quite reasonable to migrate to popular Harvard architectures (e.g., AVR, 8051, PIC, etc.), many modern 32-bit processors tend to be Von Neumann and while many popular architectures in this category have hardware ESP support (e.g., ARMv6+, MIPS32r3+) there are others which do not have such support. Considering that older Von Neumann processors will continue to be produced and integrated into new systems, this will leave a segment of embedded devices without hardware ESP support which is an open problem. Additionally, existing software ESP solutions (e.g., PnX’s NOEXEC) only support a limited number of OS and architecture combinations.
As such, low-overhead software ESP support for a wide range of common embedded operating systems and processor architectures is currently an open problem. Additionally, while Table IV shows that the majority of embedded operating systems offer memory protection support, not all embedded hardware offers the required underlying features to allow the OS to make use of this support. Table VII shows only 47% of all surveyed core families have MMU support and only 12% have MPU support, which leaves 41% unable to accommodate memory protection. Due to cost sensitivity as discussed in Section IV-A embedded systems manufacturers are unlikely to migrate to costlier higher-end processors with MPU/MMU support mainly for security reasons and as such this leaves us with the open problem.

2) Virtual Memory: As discussed earlier, real-time requirements and lack of MMU support adversely affect embedded virtual memory adoption. While Table IV shows that 50% and 44% of all analyzed systems and all non-mobile embedded operating systems offer virtual memory support, this drops to a mere 19% if we eliminate the Linux-, BSD- and Windows-based ones. When it comes to deeply embedded systems, virtual memory support is absent altogether. One also needs to take into account that even if an embedded OS offers virtual memory support, disk-less embedded systems cannot use this to extend RAM since this would requiring swapping to disk. All these constraints are so intrinsically tied to the embedded space that it is highly unlikely that we will see universal virtual memory adoption and as such the lack of alternatives to ASLR suitable for embedded systems without virtual memory remains an open problem.

3) Advanced Processor Features: Many modern security measure proposals rely on advanced processor features to offset otherwise unacceptable overhead penalties. Such features range from support for trusted computing (e.g., ARM TrustZone), complication of kernel-mode exploitation, isolation of code and data regions in memory and pointer bounds checking to features utilized to support Control-Flow Integrity (CFI) as well as cryptographic hardware acceleration.

When it comes to embedded systems, the problem with security measures which rely on such advanced processor features is that they are only available on the newest and most high-end architectures. Even among the more high-end embedded-oriented processors such as the Intel Atom or the ARMv8-based CPUs the vast majority of these features is unsupported. Additionally, such advanced processor features are not likely to be adopted by any embedded-oriented processors other than the most high-end ones anytime soon either, considering the corresponding cost increase. As such, any proposal for embedded security measures seeking widespread adoption will need to avoid relying on such advanced processor features.

4) OS CSPRNGs: Secure randomness plays a fundamental role in the wider security ecosystem, not only for cryptographic purposes but also as a dependency upon which exploit mitigations rely. Since the design and implementation of a CSPRNG is not a trivial affair, the provision of secure randomness can be considered an important OS service. But as can be seen in Table IV OS CSPRNG support is far from universal in embedded operating systems. This is particularly visible in the non Linux-, BSD- and Windows-based operating systems and even more so in those aimed at deeply embedded systems. Porting existing OS CSPRNG designs from the general-purpose world to the embedded world, even if it is from a GP-oriented version to an embedded-oriented version of the same operating system, is far from trivial for various reasons which we describe in the following.

a) OS & Hardware Diversity: As discussed earlier in this work, the embedded world is heavily fragmented. The fact that embedded operating systems often seek to cater to platforms with much more divergent capabilities than their general-purpose counterparts means it is hard to identify universally available, suitable entropy sources. So while there exists a sizeable body of work around the design of embedded random number generators, these designs are generally very domain-specific as they rely on entropy sources (e.g., sensor values [26], radio and GPS data [27]) present only in specific embedded devices.

b) Resource Constraints: The resource constraints discussed in Section IV-B also impact embedded PRNG design. Limited processing power, memory and code size constraints translate to a need for lightweight cryptography [28]; small, fast algorithms which still offer the appropriate degree of security. In addition, power consumption constraints necessitate a PRNG design that avoids constant entropy collection, especially considering many battery-operated deeply embedded devices spend most of their time in standby modes waiting for event- or time-based activation to preserve battery life.

c) Low Entropy Environment: Perhaps the biggest hurdle in embedded PRNG design is the fact that embedded systems are generally a low entropy environment. Since they are designed for specific, limited tasks. In the general purpose world, where one can assume most systems have user peripherals and disks one can use the associated system events (e.g., keystroke timings, mouse movements) as a source of entropy. But for most embedded systems, being headless and/or disk-less as well as having no user interaction, this is not an option.

Ideally, this problem would be solved by having omnipresent, on-chip True Random Number Generator (TRNG) available but considering embedded cost sensitivity issues this is not realistic. So in practice, one sees a lot of workarounds of dubious quality, which tend to lead to security issues of their own. Common and insecure approaches are to use personalization data (e.g., device MAC addresses or serial numbers) as seed entropy [29] or rely on manufacturer-supplied initial entropy, sometimes in the form of a so-called seed file. But care needs to be taken here that these seed files are unique per device, unpredictable and secret. This approach still leaves various problems for embedded systems such as dealing with disk-less nodes and not being applicable to the first system boot (which is often when embedded devices generated their long-term cryptographic keys).
V. SUMMARY OF OPEN PROBLEMS

Based on the mentioned constraints and our quantitative results, we can identify two pressing open problems relating to embedded exploit mitigation adoption namely: exploit mitigation and OS CSPRNG design for deeply embedded systems.

A. Deeply Embedded Exploit Mitigation Criteria

We can distill the following criteria for deeply embedded exploit mitigations based on the observations in Section IV:

1) **Limited Resource Pressure**: Mitigations should limit pressure on constrained resources to a minimum and provide low worst-case (rather than average-case) overhead upper bounds. As observed by Szekeres et al. [30], exploit mitigations are only likely to see widespread industry adoption if the average-case imposed code size, memory and runtime performance overhead is between at most 5 and 10%.

2) **Hardware Agnostic**: Mitigation designs should be hardware agnostic to widen deployability across the embedded hardware. This rules out any dedicated hardware proposals and any reliance on specific hardware features that are not commonly available in deeply embedded systems. This does not include hardware features commonly but not universally available such as hardware ESP.

3) **Availability Preservation**: Mitigations should offer multiple measures to take upon detection of an attack that allows for different degrees of availability preservation, ranging from those that allow an attack to take place without interfering to those that reduce availability disruption to a minimum. The rationale behind the former is that if availability is of prime importance, the worst-case scenario for an exploited vulnerability is to disrupt this availability and as such an unhindered but reported attack that gains control of the system and keeps it up is preferable over a prevented attack that brings it down in the process.

4) **Real-Time Friendly**: Mitigations should not violate real-time requirements and as such avoid non-deterministic constructs. As discussed earlier, this rules out designs relying on virtual memory.

5) **Easy (RT)OS Integration**: Mitigations should be easy to integrate into existing (RT)OS without requiring significant redesign of the operating system itself to widen deployability across the embedded OS and reduce integration cost.

B. OS CSPRNG Design for Deeply Embedded Systems

We can distill the following criteria for non-domain specific deeply embedded OS CSPRNGs based on the observations we made in Section IV:

1) **Lightweight Cryptography**: The CSPRNG will have to be based on lightweight cryptographic primitives [28] to accommodate code & data memory as well as processing power constraints. Any OS CSPRNG design targeting deeply embedded systems should be deployable on a representative hardware platform and only utilize a small fraction of available resources.

2) **Entropy Gathering Limitations**: The CSPRNG will have to be designed in such a way as to not rely on constant runtime entropy gathering to reduce power consumption. This means entropy collection will have to be rapid and preferably take place mostly during system startup, given that many battery-operated embedded systems are in standby or powered off between small periods of event- or time-triggered activity.

3) **Non-Domain Specific Entropy Sources**: The CSPRNG will have to draw upon entropy sources that are both suitable in terms of entropic quality as well as nearly universally present in deeply embedded systems. Ideally, such entropy sources have high throughputs so sufficient entropy is rapidly available at system startup and runtime entropy gathering can be limited. While there is nothing preventing CSPRNG augmentation with additional platform- or device-specific entropy sources (e.g., sensor values), these should not be the primary sources nor should the choice of entropy sources be left up to the system integrator.

VI. \( \mu \text{Armor Design} \)

In this section, we propose \( \mu \text{Armor} \), an exploit mitigation and OS CSPRNG baseline design for deeply embedded systems in the form of LLVM passes. \( \mu \text{Armor} \) seeks to address the relevant gap areas based on the results from our quantitative analysis. \( \mu \text{Armor} \) is targeted at those deeply embedded systems which satisfy the following conditions:

- Feature either a (modified) Harvard architecture CPU or Von Neumann one with an MPU with hardware ESP support.
- Run a low-end deeply embedded OS (e.g., Zephyr, FreeRTOS, TinyOS) or kernel with a single address space and without virtual memory support. The OS is allowed to be multiple-stack, multi-threading, and real-time capable.

A. Attacker Model

We assume an attacker who is capable of exploiting memory corruption vulnerabilities within the target deeply embedded OS. Attacker wants to use such vulnerabilities to execute arbitrary code or invoke arbitrary system functionality. We assume that the attacker attempts to exploit a vulnerability over a networking protocol (e.g., Ethernet or WiFi). We assume the attacker does not have access to the specific firmware image of the target device, but she may have access to the firmware image of another instance of the system. Finally, \( \mu \text{Armor} \) does not seek to protect against data-flow hijacking or data-only attacks.

B. High-Level Design

\( \mu \text{Armor} \) incorporates three mitigations measures in order to match the functionality of the baseline outlined in Section III \( \mu \text{ESP}, \mu \text{Scramble}, \) and \( \mu \text{SSP} \) in the form of LLVM passes.
In addition, it includes µRNG in order to provide required OS CSpRNG support.

C. µESP Design

µESP is the Executable Space Protection (ESP) component of µArmor and unlike other ESP implementations is explicitly designed for MCUs running single address space OSes. µESP assumes the OS can be modified for ESP-compliance, i.e., it allows for separation of code and data memory regions as well as avoiding code constructs such as dynamically generated code, stack-stored trampolines, etc. µESP explicitly sets the hardware ESP non-executable bit for every memory region belonging to a non-code region, while not setting it for those owned by code region. Also, it ensures that no write permissions are set for memory regions belonging to a code region to avoid code modification attacks.

Furthermore, dynamic data memory objects (e.g., stack and heap) are ensured to be fully placed in a data memory region, and the stack is instructed to grow away from other data regions. Since most stacks grow downward, by placing the stack at the bottom of data memory, an overflow will cause an exception either because it is a code region in RAM (thus caught by µESP permissions) or because we try to write outside of RAM. In a multi-stack environment, individual stacks overflowing into each other could be captured by placing a guard at the end of each stack if MPU granularity and region count allows for this.

Finally, after setting up permissions, µESP will mark any code regions responsible for MPU interaction or flash rewriting (e.g., in the bootloader) as non-executable. The reason behind such change is to avoid code-reuse attacks targeting these regions for permission-changing payloads or ret2bootloader attacks [31], [32], [33] and will disable further changes to the MPU by making the relevant control registers non-writable.

D. µScramble Design

µScramble is a compile-time code diversification scheme which imposes minimal runtime overhead. Compile-time diversification allows us to leverage the high-level information available to the compiler, target multiple hardware platforms implicitly, avoid the need for disassembly and binary analysis as well as operate in an automatic fashion (as part of the regular compilation process) transparent to software developers.

Note that the choice between diversification at compile-time requires infrastructures from the vendor. However, we believe that making µScramble as a compile-time code diversification is reasonable due to results discussed on Section [IV] and major resource constraints exist within deeply embedded systems.

Since the goal of µScramble is to thwart code-reuse attacks, we aim to either eliminate gadgets, randomize them or make it infeasible to guess their location in memory. µScramble seeks to achieve this by diversifying code in a fine-grained manner at compile-time. Additionally, since µScramble needs to be semantics-preserving and to take into account embedded resource constraints regarding code size, memory usage and performance overhead we have created the diversification transformations listed below for µScramble:

a) Register-Preservation Reordering: Most architectural calling conventions specify which registers are callee-saved and which are considered scratch registers. Compilers take note of all registers used within a given subroutine and will ensure that those which are callee-saved are stored to the stack during the function prologue and restored from it during the epilogue. Such sequences are one of the most common targets for code-reuse gadgets due to their ability to act as register-setters terminated by a return instruction. Since the exact order in which these registers are saved to and restored from the stack does not matter, we can randomize it and thus break gadget chain assumptions about what values end up in what registers.

b) Dead Code Insertion: µScramble supports two types of dead codes namely, no-operation (NOP)-equivalent instructions that do not present opportunities for unaligned instruction gadgets and trap instructions which activate a violation policy handler upon execution (something which never happens during regular execution). The latter has the benefit of raising alerts while any attempt to brute-force gadgets is in progress.

c) Function Reordering: Unlike common function re-ordering techniques [34], here due to embedded systems limitation, we randomize only the function order and as such the degree of diversification introduced is determined by the number of functions present in the target code.

The above transformations affect both code topology and code itself by randomizing the offsets of a) instructions with respect to a function address, b) functions with respect to the image base and c) one function with respect to another function as well as randomizing the order of register preservation code. Due to the fine-grained nature of our diversification we reduce memory object correlation and offer better protection against information leaks and brute-force attacks than coarse-grained schemes.

E. µSSP Design

µSSP is a component of µArmor which ensures proper separation of data and pointers within a given local stackframe. µSSP works by placing the latter below the former so that stack overflows cannot target code or data pointers residing on the stack while the stack canary shields the stackframe metadata (e.g., saved frame pointer, return address). µSSP also complies with regular GCC Stack Smashing Protection (SSP) function coverage parameters and is capable of protecting all kernel- and application-code that runs after early kernel and C support initialization.

On the operating system side, µSSP uses a single master canary generated once at system boot for all OS tasks and threads. Since on deeply embedded systems without virtual memory there is no memory isolation for OS tasks nor a separation between kernel- and userspace, periodic canary renewal would lead to synchronization conflicts for a shared canary. Our solution for this problem would be assigning a
dedicated master canary for each thread (or possibly only OS tasks) as well as one for the kernel and renewing canaries upon thread startup. The problem here, however, is that without virtual memory different threads utilize shared code which would require the compiler to figure out which code is used exclusively by a given thread and which code is shared, assigning a single common master canary for all shared code to prevent synchronization problems. This limits renewal effectiveness to such a degree, especially compared to incurred overhead cost, that we simply opt for the single master canary approach. In addition to that, the single address space nature (and accompanying lack of privilege separation) of most deeply embedded OSes would render a multi-canary scheme rather irrelevant as well.

As far as canary generation is concerned, \( \mu_{SSP} \) assumes the presence of either an OS CSPRNG (e.g., \( \mu_{RNG} \) described in Section VII-D) or a TRNG (True Random Number Generator). We provide OS CSPRNG support as part of \( \mu_{RNG} \).

The final components of \( \mu_{SSP} \) is its modular canary violation handlers. Deeply embedded exploit mitigations should offer multiple courses of action to be taken upon attack detection to allow for different degrees of availability preservation. We provide (in case the OS do not support it) different type of handlers which are discussed in Section VII-D.

F. \( \mu_{RNG} \) Design

\( \mu_{RNG} \) is our modification of a compact, software-only CSPRNG design for ARM Cortex-M by Van Herrewege et al. \[35\] with a 128 bit security strength level. It utilizes the lightweight Keccak \[36\] sponge function as a CSPRNG \[37\]. For obvious reasons, we do not create our own CSPRNG function. However, \( \mu_{RNG} \) is not a simple reimplementation of Keccak \[35\]. \( \mu_{RNG} \) is extended to function as an OS CSPRNG by adding reseed control suitable to constrained embedded systems. More specifically, it avoids the constant runtime polling for reseeding, typical in most OS CSPRNG designs which puts a strain on power consumption. The main purpose of \( \mu_{RNG} \) in the context of this work is to serve as a dependency for exploit mitigations (such as stack canary mechanisms) but depending on chosen security strength, it is perfectly suitable as a general-purpose CSPRNG. While, for the sake of convenience, this work describes and implements \( \mu_{RNG} \) in the context of our representative platform, the \( \mu_{RNG} \) design is not restricted to any OS or platform in particular.

In \( \mu_{RNG} \) entropy accumulation is done by Keccak sponge absorption functionality while random number generation is done by squeezing the Keccak sponge, allowing us to use the same algorithm for both purposes. \( \mu_{RNG} \) uses the Keccak-f[200] permutation with rate and capacity parameters \( r = 64 \) and \( c = 136 \) respectively. Note that \( \mu_{RNG} \) is fully reseeded every 1GB of output which results in a Keccak internal state of 25 bytes and generation of 64-bit pseudo-random numbers per squeeze operation. We initially seed \( \mu_{RNG} \) with at least 256 bits of entropy and ensure reseeding is done with at least 256 bits of entropy.

When designing reseed control we need to take into account the applicability of passive and active state recovery attacks \[37\]. In case of the former, the attacker cannot influence seed data while in case of the latter the attacker can. As per the original design by Van Herrewege et al. \[35\] upon which \( \mu_{RNG} \) is based, \( \mu_{RNG} \) provides the required security against passive state recovery attacks as long as reseeding occurs at least every \( r \ast 2^r = 64 \ast 2^{32} = 32GB \) of PRNG output and against active attacks as long as reseeding happens at least every \( 2^8 \ast r = 256 \ast 64 = 2KB \) of output. Since our attacker model explicitly assumes a remote attacker, incapable of influencing our entropy sources remotely, we only take passive state recovery attacks into account. We have to consider a trade-off between overhead and security with respect to reseed frequency; ideally reseeding is done regularly to keep as much entropy in the PRNG as possible at all times but frequent entropy gathering puts pressure on embedded resources in terms of memory and power consumption. \( \mu_{RNG} \) has two options for reseed control:

- **Consistent**: Reseed control here is integrated into the PRNG output function, ensuring at least 1 bit of entropy is accumulated for every 64 bits of PRNG output, thus ensuring a full 256-bit reseed every 2KB of output.
- **Periodic**: Reseed control is integrated into the PRNG output function as well, together with a 32-bit reseed counter which keeps track of the number of bytes output, but actual reseed functionality is only invoked after the counter exceeds a certain threshold value \( T \). Reseed functionality is designed to run for at most \( S \) seconds (to facilitate worst-case timing estimates) and accumulate entropy while resetting the reseed counter.

Additionally, \( \mu_{RNG} \) uses non-domain specific entropy sources that can be found on most embedded devices. We can divide these sources into two groups:

- **Initialization**: Initial entropy is gathered during early boot and should be rapidly available in sufficient quantity upon system startup to avoid the so-called boot-time entropy hole \[38\]. We follow \[35\] in using SRAM Startup values (SUVs) as our primary source of initial entropy. Using SRAM SUVs as a source of initial entropy allows us to have an entropy source that is present on most embedded devices, instantly available in (very) early boot and differs from boot session to boot session as well as from device to device. As discussed in the literature \[39\], \[40\], \[35\], the amount of entropy in modern microcontroller SRAM tends to be around 5% of its total size at normal operating temperatures. This means that, on average, \( \mu_{RNG} \) would require at least \( 2 \ast \frac{128}{0.05} = 640 \) bytes of SRAM to guarantee a security strength level of 128 bits, a reasonable restriction for most modern microcontrollers.

- **Reseeding**: Reseed entropy is gathered upon invocation of reseed control functionality and should provide either at least 1 bit of entropy per invocation (in case of consistent reseed control) or an appropriate throughput rate (in case of periodic reseed control).
TABLE VIII: μESP Memory Permission Policies

| Memory          | Perms. |
|-----------------|--------|
| Code (sensitive)| RO+XN  |
| Code (other)    | RO+X   |
| Data            | RW+XN  |
| Peripherals     | RW+XN  |
| SCB Config      | RO+XN  |
| MPU Config      | RO+XN  |

- **Clock Jitter & Drift:** The various oscillators (e.g., RC, Ring or VC oscillators) acting as microcontroller clock signal sources are never completely stable and are influenced by factors such as supply voltage, temperature, etc.
- **ADC Noise:** Many embedded systems are outfitted with analog-to-digital converters (ADCs) we can use it as an entropy source by sampling the least significant bit of ADC output corresponding to floating inputs. It is worth mentioning that general suitability of ADC Noise as cryptographic entropy source is unevaluated and as such we recommend against integration unless proper device-specific evaluation indicates suitability.

VII. μARMOR IMPLEMENTATION

In the following, we describe implementation aspects for all μArmor components.

A. Representative Platform

μArmor is not intrinsically tied to any particular OS or hardware configuration. However, for our implementation we choose Zephyr [41] RTOS on a TI LM3S6965 microcontroller, which is based on a 50 MHz ARM Cortex-M3 outfitted with 256 kB flash, 64 kB SRAM and an MPU. We chose Zephyr because it is an actively developed, open-source OS with a permissive license aimed at resource-constrained embedded devices and is supported by the Linux Foundation and major chip vendors such as Intel, NXP, and Synopsys. We picked the TI LM3S6965 microcontroller because it is supported by Zephyr and is representative of a typical deeply embedded system with limited resources. As noted above, the μArmor protection scheme is implemented as LLVM passes.

B. μESP Implementation

We consider the following two common approaches to deal with code and data memory in embedded systems: (i) program code is located in and executed from flash and data is copied to RAM and (ii) both program code and data are copied from flash to RAM by a first stage bootloader and further code (e.g., the OS kernel) is executed from RAM. For μESP this corresponds to the permission policies outlined in Table VIII. The sensitive code is a code which handles rewriting flash memory, copying data from flash to RAM and setting up memory permissions and is made non-executable after execution. The MPU Config refers to MPU configuration registers which are made read-only after memory permissions have been set up and the SCB config refers to the System Control Block (SCB) configuration registers.

With the Cortex-M3's MPU available on TI LM3S6965 we can enforce μESP policies by making use of its support for up to 8 memory regions. Memory regions can cover the full 4 GB address space and come with size and permission (in the form of XN and data access flags) attributes. Memory regions start addresses must be size-aligned (ie. a 2 KB region must start at an address that is a multiple of 2 KB). We do not make use of the available privilege modes because we do not want to assume OS compatibility with them and as such our permissions apply to both privileged and unprivileged modes. Based on the policies in Table VIII we construct a μESP configuration for the TI LM3S6965 in Tables IX and X representing settings for execute-from-flash and RAM code relocation scenarios respectively.

We start with a default region, using the lowest region number, which covers the entire address space with RW+XN permissions. If two memory regions overlap on the MPU, region attributes fall back to the region with the highest region number. We can use this feature to limit the number of regions we have to specify and define overlapping regions for exceptions to default data memory. SRAM and peripherals are covered by this default region as well. We define a region for code (covering all of flash memory) with RO+XN permissions and use a higher region as an XN overlay for any sensitive code (except the final line which locks the MPU) to be made non-executable after system initialization.

For the scenario where code is executed from RAM, we provide identical regions to be placed wherever in RAM the bootloader relocates code to. Note that since aliased address ranges need to be covered by the same permissions, this might mean memory regions could need sizes bigger than the actual amount of on-chip SRAM.

In addition to the above we have to consider the following security-sensitive memory regions: Interrupt Vector Table (IVT) and System Control Block (SCB). The IVT holds exception vectors such as the stack pointer reset value and start address (loaded upon system reset) as well as interrupt handler addresses. The IVT would be an interesting overwriting target for attackers but by default it lives completely within the lower
region of flash memory and as such is covered by the RO+X permissions of our code region. It is possible, however, to relocate the vector table using the Vector Table Offset Register (VTOR) in the System Control Block (SCB). If the vector table is relocated to RAM along with other code as part of a bootloader, this is not an issue because it will be covered by the relevant code region. But to protect the system from forcing a malicious relocation as part of an exploit (among other things), we mark the SCB as read-only. If some SCB functionality should be writable during runtime, \( \mu \)ESP uses the MPU’s sub-region feature which divides a region into 8 equally large sub-regions (provided the region size is at least 256 bytes) that can be disabled individually thus falling back to default RW+XN permissions. If so desired, one could merge the SCB and MPU memory regions into a single region using disabled sub-regions to cover any addresses within the range which should have different permissions.

C. \( \mu \)Scramble Implementation

The \( \mu \)Scramble diversification transformations were implemented as LLVM passes which we describe in the following.

a) Register-Preservation Reordering: This transformation is implemented as a MachineFunctionPass which obtains the callee-saved registers of a function using `getCalleeSavedRegs`, shuffles their order using the LLVM PRNG and sets the new order.

b) Dead Code Insertion: This transformation is implemented as a MachineFunctionPass which identifies the final basic block of a given function and generates dead code-stubs. We allow developers to specify what type of dead code-stub (`NOP or trap`) they wish to generate with a compiler flag. NOP-stubs consist of a single, repeated, architecture-dependent `NOP` instruction to ensure minimal gadget usefulness.

Trap-stubs consist of branch instructions to a violation policy handler, in our case we use the same handler used for \( \mu \)SSP violations described below.

c) Function Reordering: This transformation is implemented as a ModulePass which retrieves the current modules function list and shuffles it using the LLVM PRNG. The linker will ensure functions are organized in the randomized order in the produced firmware image. All randomization operations used in \( \mu \)Scramble draw upon the LLVM PRNG which draws upon a developer-supplied true random seed. Since this PRNG is deterministic this means a given firmware build can be reproduced from the seed as is done by Gionta et al. [42].

D. \( \mu \)SSP Implementation

We implemented \( \mu \)SSP as an augmentation of Zephyr’s SSP implementation. Since Zephyr uses the GCC SSP model it already meets \( \mu \)SSP’s compiler-side criteria. On the OS side, it stores a single master canary value as a global variable in .bss and initializes it at boot (as part of the _Cstart function, after hardware initialization but before the main thread is activated) by drawing from the `sys_rand32_get` API.

We augmented this SSP implementation by adding support for a terminator-style canary bitmask, ensuring an OS CSPRNG is available for secure canary generation and implementing a modular canary violation handler.

- **Passive**: The violation handler simply returns to the violating function.
- **Fatal**: The violation handler try to terminate the violating thread and continue running the system.
- **Thread Restart**: To properly handle thread restarts we maintain a global list (a hash table) of thread restart handlers associated with thread IDs. We require the thread ID be registered together with the restart handler upon thread start and de-registered upon thread termination. Upon invocation of the violation handler, the violating thread’s ID is looked up in the restart handler list, the associated restart handler is fetched, the thread in question is terminated and the restart handler is invoked.
- **System Restart**: We invoke the `sys_reboot` API with the `SYS_REBOOT_COLD` argument to perform a system restart.
- **System Shutdown**: This approach depends on SoC power management subsystem implementations. In the absence of such functionality we default to terminating all running threads.

E. \( \mu \)RNG Implementation

We implemented \( \mu \)RNG as a driver for the Zephyr random API. \( \mu \)RNG output can be requested with the `sys_rand32_get` API which squeezes the \( \mu \)RNG keccak output as a driver for the ARM Cortex-M's vector table. Note that almost all RTOSe provide such initialization routine. This ensures SRAM is untouched before our \( \mu \)RNG initialization code is invoked.

VIII. \( \mu \)ARMOR EVALUATION & LIMITATIONS

A. Overhead Evaluation

We evaluate the overhead imposed by \( \mu \)Armor in terms of code size, data size, memory usage, and runtime increases. Code and data size figures represent increases in code (in flash) and constants (in SRAM) respectively. Memory usage increases represents a worst-case SRAM overhead imposition (by use of dynamic data structures) at any point during execution. We instrumented application code to measure runtime performance using a hardware high precision counter. Runtime performance figures represent increases in the number of
clockcycles consumed for a given amount of code to run, reported as the average of 25 runs.

We evaluate μESP, μSSP, μRNG separately in Tables XII XIII and XIV and μScramble in Tables XIX. Note that evaluation results for memory overhead is based on worst-case estimate and the results for runtime overhead are from average of 25 runs. To get an idea of the overhead on realistic applications we chose three sample Zephyr IoT applications stressing different subsystems: Additionally, due to space restrictions, we eliminated in Table XIX the results for the data size, memory, and runtime overhead with respect to application and runtime since there was no difference before and after using μScramble.

- **philosopher**: An implementation of the dining philosophers problem using multiple preemptible and cooperative threads of differing priorities [43].
- **net/echo_server**: An IPv4/IPv6 UDP/TCP echo server application [44].
- **net/telnet**: IPv4/IPv6 telnet service providing a shell with two shell modules: net and kernel [45].

| TABLE XI: μScramble Overhead with respect to Applications (A) and Resources (R) in average of 25 variants. |
|----------------------------------------------------------------------------------------------------------|
| **App** | % CS (A) | % CS (R) | **App** | % CS (A) | % CS (R) |
|------------------------------------------------------------|-----------|-----------|------------------------------------------------------------|-----------|-----------|
| Application | | | Test | 12 | 0 | |
| powerwindow | 2.2 | 0.1 | duff | 3 | 0 | 0.1 |
| test3 | | | | | | |
| Kernel | | | | | | |
| Binarysearch | 3.3 | 0 | addpcm_dec | 11 | 0 | |
| bitcount | 2.3 | 0 | addpcm_enc | 1.3 | 0 | |
| bitonic | 4.2 | 0 | armagam | 1.3 | 0.1 | |
| bsort | 3.5 | 0 | anagram | 1.8 | 0 | |
| complex_update | 1.6 | 0 | audiobeam | 1.6 | 0 | |
| countnegative | 3.4 | 0 | cjpeg_transpuuç | 1.2 | 0 | |
| fac | 4.0 | 0 | dijkstra | 1.9 | 0 | |
| fft | 2.6 | 0 | epic | 1.6 | 0 | |
| filterbank | 1.0 | 0 | fnref | 1.0 | 0 | |
| fir2dim | 1.5 | 0 | gm_dec | 1.3 | 0 | |
| lir | 2.2 | 0 | h264_dec | 0.8 | 0 | |
| insertsort | 1.9 | 0 | huff_dec | 1.8 | 0 | |
| jf dctint | 1.3 | 0 | huff_enc | 1.6 | 0 | |
| lms | 1.4 | 0 | mpeg2 | 1.1 | 0.1 | |
| ludpmsg | 0.9 | 0 | ndes | 1.0 | 0 | |
| matrixl | 3.0 | 0 | petrinet | 0.2 | 0 | |
| random | 1.9 | 0 | r1jnda1_dec | 0.3 | 0 | |
| minver | 0.8 | 0 | r1jnda1_enc | 0.3 | 0 | |
| pm | 0.8 | 0 | statemate | 0.5 | 0 | |
| prime | 1.8 | 0 | | | | |
| quicksort | 0.9 | 0 | | | | |
| recursion | 4.6 | 0 | | | | |
| sha | 1.8 | 0 | | | | |
| st | 2.0 | 0 | | | | |
| basicmath | 0.7 | 0 | | | | |

| Percentage of Code Size (CS) Overhead |

We evaluate μESP, μSSP, and μRNG against the above representative applications compiled for the TI LM3S6965 with GCC as provided by the Zephyr SDK. We evaluate μScramble against a different set of applications since the overhead imposed by μScramble on a single application is non-deterministic and is dependent to parameters such as the number of functions. Thus we evaluate μScramble against a set of 50 benchmarks and applications from the TACLeBench suite [44]. TACLeBench consists of self-contained programs without external or OS dependencies and is drawn from well-known (embedded) benchmarking suites such as DSP-Stone [47], MRTC WCET [48], SNU-RT [49], MiBench [50], MediaBench [51], NetBench and HPEC [52]. The benchmarks in question are drawn from various embedded domains ranging from automotive and networking to security and telecommunications and are sub-divided into application, kernel, sequential and test groups implementing realistic applications, small kernel functions, large sequential functions and artificial stress tests respectively.

| TABLE XII: μESP Overhead Evaluation with respect to (wrt) application and resources |
|----------------------------------------------------------------------------------------------------------------------------------|
| **Wrt. Application** | % Code | % Data | % Memory | % Runtime |
| philosopher | 1.5 | 0 | x (0 B) | 0 |
| echo_server | 0.2 | 0 | x (0 B) | 0 |
| telnet | 0.2 | 0 | x (0 B) | 0 |
| Wrt. Resources | | | | |
| philosopher | 0 | 0 | x | |
| echo_server | 0 | 0 | x | |
| telnet | 0 | 0 | x | |

| TABLE XIII: μSSP Overhead Evaluation |
|-------------------------------------------------------------------------------------|
| **Wrt. Application** | % Code | % Data | % Memory | % Runtime |
| philosopher | 26.4 | 0 | x (84 B) | 0.7 |
| echo_server | 14.1 | 0.1 | x (52 B) | 0 |
| telnet | 27.3 | 0 | x (84 B) | 0.7 |
| Wrt. Resources | | | | |
| philosopher | 0 | 0 | x | |
| echo_server | 0 | 0 | x | |
| telnet | 0 | 0 | x | |

| TABLE XIV: μRNG Overhead Evaluation |
|-------------------------------------------------------------------------------------|
| **Wrt. Application** | % Code | % Data | % Memory | % Runtime |
| philosopher | 10.2 | 0.4 | x (50 B) | 0 |
| echo_server | 14.1 | 0.1 | x (52 B) | 0 |
| telnet | 1.5 | 0.1 | x (52 B) | 0 |
| Wrt. Resources | | | | |
| philosopher | 0 | 0 | x | |
| echo_server | 0 | 0 | x | |
| telnet | 0 | 0 | x | |

We are not interested in average memory usage overheads but rather in worst case figures because of potentially unacceptable SRAM pressure. Using stack depth analysis embedded developers get an indication of the maximum amount of memory used by the stack in their application. We have decided to obtain a stack depth estimate in between the usual lower bounds derived from experimental observation and the upper bounds derived from static analysis. This estimate is derived from multiplying the longest identified call chain in the program Control Flow Graph (CFG) by the overhead imposed by a single canary. Note that we report overhead figures both with respect to the original unprotected application and with respect to total device resources.

Based on the reported figures above, we can conclude code size overheads stay below 5% with respect to the application for all components except μSSP and μRNG and are less than or equal to 5% with respect to total device resources for all components. Data size, memory usage and runtime overheads all stay well below 1% both with respect to the application as well as with respect to total device resources. μSSP code
size overheads are clearly the heaviest overhead imposition of all metrics and components. μSSP introduces roughly 4 instructions of prologue and 5 instructions of epilogue overhead amounting to 36 bytes per protected function. While the average code overhead with respect to the unprotected application is 28.1%, we can see overhead in terms of total resource pressure remains equal to or below 5%.

### B. Security Evaluation and Limitations

1) μESP Security: μESP protects against both code injection and code modification by enforcing a separation between code and data memory, forcing an attacker to use a code-reuse payload. By locking the MPU and rendering μESP and bootloader code non-executable after it has been run, μESP protects against code-reuse attacks that seek to circumvent μESP by means of permission-changing payloads or ret2bootloader attacks [32, 33] that seek to rewrite flash memory with attacker-injected code.

2) μScramble Security: To get an idea of the entropic quality of μScramble transformations we performed a coverage analysis consisting of taking our selection from the TACLEBench suite and generating 1000 different μScramble-diversified variants for each benchmark. We then harvest all gadgets from each variant using a ROPGadget [53] and determined, for each gadget in each variant, in how many iterations the gadget survives. These gadget survival rates give us an indication as to the quality of μScramble’s coverage of the target gadget space. The detailed results of this analysis are reported in Table XV.

| Application          | Avg. GS² | Max. GS² | Application          | Avg. GS² | Max. GS² |
|----------------------|----------|----------|----------------------|----------|----------|
| lift                  | 2.1      | 2.1      | code                 | 29.8     | 143.0    |
| powerwindow          | 12.0     | 12.0     | duff                 | 1.6      | 7.0      |
| Kernels              |          |          | test3                | 0.0      | 0.0      |
| Sequential           |          |          |                      |          |          |
| βinjurysearch        | 1.4      | 1.4      | apcm_enc             | 0.5      | 2.0      |
| bitcount             | 63.3     | 180.0    | ammunition            | 0.6      | 5.0      |
| bitonic              | 85.1     | 166.0    | ammunition            | 3.6      | 147.0    |
| bsort                | 58.5     | 171.0    | anagram               | 0.6      | 4.0      |
| complex_upda         | 68.1     | 199.0    | audiobeam             | 4.0      | 79.0     |
| countnegativ         | 2        | 11.0     | cjpeg_trans           | 32.0     | 96.0     |
| fac                  | 2.5      | 4.0      | cjpeg_wrbmp           | 1.4      | 3.0      |
| fft                  | 1.2      | 5.0      | dijkstra              | 1.3      | 5.0      |
| filterbank           | 59.9     | 209.0    | epic                  | 0.0      | 0.0      |
| fir2dim              | 70.5     | 210.0    | fmsref                | 0.5      | 3.0      |
| firl                 | 2.5      | 8.0      | gsm_dec               | 1.3      | 43.0     |
| insertsort           | 101.5    | 202.0    | h264_dec              | 100.0    | 181.0    |
| jfdctint             | 97.0     | 193.0    | huff_dec              | 18.8     | 91.0     |
| lema                 | 64.2     | 193.0    | huff_enc              | 5.8      | 55.0     |
| ludcpl               | 55.4     | 166.0    | mpdec                 | 0.0      | 0.0      |
| matrixl              | 68.8     | 203.0    | ndes                  | 0.6      | 2.0      |
| md5                  | 0.6      | 7.0      | petrinet              | 2.0      | 2.0      |
| minver               | 1.3      | 5.0      | rijnjael_de           | 2.5      | 16.0     |
| pm                   | 12.5     | 97.0     | rijnjael_en           | 3.0      | 11.0     |
| prime                | 0.9      | 5.0      | statement             | 16.4     | 97.0     |
| quicksort            | 1.5      | 9.0      |                      |          |          |
| recursion            | 91.6     | 188.0    | sha                   | 2.3      | 6.0      |
| sha                  | 3.8      | 3.0      | stat                   | 0.8      | 3.0      |
| basicmath            | 1.4      |          |                      |          |          |

¹ Gadget Survival

The results of our analysis demonstrate that the highest average gadget survival rate is 101.5 (for insertsort) and the highest maximum gadget survival rate is 210 (for fir2dim) while most remain well below those numbers. This means that in a worst-case scenario, a single gadget survives across roughly 10.15% of variants on average and across 21% of them at most, requiring brute-force for all other variants. Thus an attacker constructing a code-reuse payload from a given firmware variant cannot expect any gadget in this payload to work beyond 10.15% of target devices. Attacker also needs to consider that the gadget survival variants of different gadgets do not necessarily intersect. As such, prospects for gadget chain survival are even worse and brute-force search space scales with respect to payload length as well.

3) μSSP Security: The security offered by μSSP is inherently constrained by the limits of stack canaries: they only protect against stack buffer overflows targeting stackframe metadata and not against other types memory corruption vulnerabilities nor against stack buffer overflows targeting local code or data pointers. That being said, μSSP draws upon an OS CSPRNG (in the form of μNRG) to generate canaries with 32 or 24 bits of entropy (depending on whether they are configured to be terminator style or not) which are, as such, not susceptible to either the insecure randomness issues or the system-side information leaks affecting the original Zephyr SSP canaries in the absence of a TRNG. Since the master canary value is refreshed upon system boot, the system restart, Fatal and system shutdown violation policies are not susceptible to brute-force approaches at all. The thread restart policy is susceptible to brute-force attacks. All approaches, however, raise at least an alert upon invocation. We believe that given our attacker model, 32 bits of entropy is sufficient for a brute-force attack to be infeasible.

### C. Limitations

In the following, we discuss limitations of our approach and the prototype implementation. First, μArmor requires either a (modified) Harvard or Von Neumann CPU featuring an MPU with hardware ESP support. We believe that this limitation is within reasonable bounds for our system to be considered hardware agnostic, since it only excludes older Von Neumann architectures for which there is currently no way to enforce low-overhead ESP.

Second, the μScramble component only diversifies code memory and only provides per-device diversity: it does not diversify data memory nor does it diversify on a per-boot or per-application run basis. Finally, the μNRG component requires on-chip SRAM or otherwise requires an alternative source of initial entropy.

### IX. RELATED WORK

A relevant stream of work has explored software diversification. For example, AVRAND [54] is a boot-time software only diversification scheme that randomizes binary code at a per-page granularity level. The AVRAND imposes an average code size increase of 20% and requires sizable meta-data storage.
in EEPROM. Note that AVRAND requires re-flashing flash memory upon randomization which reduces embedded device lifespan. MAVR is a boot-time diversification scheme [55] for UAV control system. However, MAVR is not hardware agnostic since it requires (costly) hardware modifications. Note that both MAVR and AVRAND require re-flashing flash memory upon randomization which reduces embedded device lifespan. Abbasi et al. [56] introduced μShield, a CFI system for embedded COTS binaries with configurable protection policies to cope with limited resources in embedded systems. However, μShield addresses high-end embedded devices. Another stream of work explored firmware integrity verification for embedded systems. For example Doppelganger [57] protects embedded systems from firmware modification via software symbiotes, a specific integrity verification trap which is invoked every time it gets executed. However, Doppelganger can only verify the static part of the firmware.

Finally, Clements et al. recently proposed EPOXY [8], to protect bare-metal deeply embedded systems. There are several key limitations to the EPOXY [8] work that set it apart from μArmor. First, μArmor targets a class of deeply embedded systems not addressed by EPOXY: those running an OS. EPOXY was designed to protect single-stack bare-metal embedded systems and is not suitable for deployment on multi-stack RTOS systems without significant re-engineering (e.g., such as with regards to scalability of safe-stack overhead, integrating privilege overlays transparently with OS mechanisms).

Second, EPOXY privilege overlay security model could be broken by an attacker returning to any legitimate call-site where the privilege elevation handler is invoked. Consider, for example, the Algorithm 1.

**Algorithm 1 EPOXY Privilege Overlay Issue**

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

Here an attacker could return to line 2 where supervisor call (SVC 0xFE) would invoke the custom EPOXY SVC_Handler (see related EPOXY code at [58]). The handler would check whether the interrupt source was correct regardless of unprivileged control-flow leading up to it and as such would elevate privileges allowing the attacker to execute the restricted operation in line 7. Code diversification would not help much because many restricted operations are located at static or easily obtainable addresses (e.g., extracted from the IVT).

Third, EPOXY code diversification approach is less fine-grained than that of μArmor since it only randomizes function order but does not diversify function sizes (and thus leaves gadget offsets with respect to function addresses intact) nor does it reorder register preservation code. As such, EPOXY code diversification is more vulnerable to information leaks than μArmor.

Finally, unlike μArmor, EPOXY does not protect against ret2bootloader style attacks [31], [33], [32] because it leaves sensitive bootloader code that cannot be relocated executable after system initialization. As a result, an attacker can return to potentially sensitive code areas allowing for re-flashing code memory or disabling the MPU.

**X. Conclusions and Future Work**

In this paper, we have presented the first quantitative study of exploit mitigation adoption in a representative selection of embedded operating systems, showing the embedded world (deeply embedded system in particular) to significantly lag behind the general-purpose world. The resulting ease of exploitation of memory corruption vulnerabilities and notoriously prolonged vulnerability exposure windows in the embedded world are cause for concern. We have presented how hardware and OS limitations and performance constraints contribute to an imposing series of constraints for developers of embedded exploit mitigations to overcome. To address this situation we have presented μArmor, an exploit mitigation baseline design for deeply embedded systems. We have shown that μArmor holds up favorably in terms of overhead imposition and offered security.

We see two main trajectories for future work on embedded binary security: long-term and short-term solutions. The former trajectory aims to develop robust techniques tackling the problem at the root and requires changes along the whole development chain. Examples are embedded-oriented safe languages and secure and scalable patching solutions.

The short-term trajectory should aim to develop solutions which reduce the impact of embedded memory corruption vulnerabilities and which can be rapidly adopted.

Finally, we should also seek for more advanced mitigations for embedded systems in order to continue raising the bar and close the mitigation security gap between general-purpose and embedded systems.

**Acknowledgment**

The work of first and third author was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 BASTION). In addition, this work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec) and the Intel Collaborative Research Institute “Collaborative Autonomous & Resilient Systems” (ICRI-CARS). The work of second and fourth authors has been supported by the NWO through the SpySpot project (no.628.001.004).

**References**

[1] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno et al., “Comprehensive experimental analyses of automotive attack surfaces.” in *USENIX Security Symposium*, 2011.
