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 $W \oplus X$ 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. [8] 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 I and II, 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.

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	Canarie	OS	ESP	ASLR	Canaries
BlackBerry OS	✓	✓	✓	Android*	✓	✓	✓
iOS*	✓	✓	✓	Win 10 Mob.*	✓	✓	✓
Sailfish OS*	✓	✓	✓	Tizen*	✓	✓	✓
Ubuntu Core*	✓	✓	✓	Brillo*	✓	✓	✓
Yocto Linux*	✓	✓	✓	Windows Embedded*	✓	✓	✓
OpenWRT*	✓	✓	✓	Junos OS*	✓	✗	✓
µClinux*	✓	✗	✓	CentOS*	✓	✓	✓
NetBSD*	✓	✓	✓	IntervalZero RTX*	✓	✗	✓
ScreenOS	✗	✗	✗	Enea OSE	✗	✗	✗
QNX	✓	✓	✓	VxWorks	✓	✗	✗
INTEGRITY	✓	✗	✗	RedactedOS [2]	✗	✗	✗
Cisco IOS	✗	✗	✗	eCos	✗	✗	✗
Zephyr	✓	✗	✓	ThreadX	✗	✗	✗
Nucleus	✗	✗	✗	NXP MQX	✗	✗	✗
Kadak AMX	✗	✗	✗	Keil RTX	✗	✗	✗
RTEMS	✗	✗	✗	freeRTOS	✗	✗	✗
Micrium µC/OS [1]	✓	✗	✗	TI-RTOS	✗	✗	✗
DSP/BIOS	✗	✗	✗	TinyOS	✗	✗	✗
LiteOS	✗	✗	✗	RIOT	✗	✗	✗
ARM mbed	✓	✗	✗	Contiki	✗	✗	✗
Nano-RK	✗	✗	✗	Mantis	✗	✗	✗

* *Embedded OS based on Windows, Linux or BSD.*
1 *A µC/OS-II kernel version with ESP support is available via a Micrium partner.*
2 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.

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.

OS	MPROT	VMEM	RNG	OS	MPROT	VMEM	RNG
Android*	✓	✓	✓	iOS*	✓	✓	✓
Win10 Mob.*	✓	✓	✓	BlackBerry OS	✓	✓	✓
Tizen*	✓	✓	✓	Sailfish OS*	✓	✓	✓
Ubuntu Core*	✓	✓	✓	Brillo*	✓	✓	✓
Yocto Linux*	✓	✓	✓	Windows Embedded*	✓	✓	✓
OpenWRT*	✓	✓	✓	Junos OS*	✓	✓	✓
µClinux*	✓	✓	✓	CentOS*	✓	✓	✓
NetBSD*	✓	✓	✓	IntervalZero RTX*	✓	✓	✓
ScreenOS	✓	✓	✓	Enea OSE	✓	✓	✗
QNX	✓	✓	✓	VxWorks	✓	✓	✗
INTEGRITY	✓	✓	✓	RedactedOS	✓	✓	✗
Cisco IOS	✗	✗	✓	eCos	✗	✗	✗
Zephyr	✓	✗	✗	ThreadX	✓	✗	✗
Nucleus	✓	✗	✗	NXP MQX	✓	✗	✗
Kadak AMX	✗	✗	✗	Keil RTX	✗	✗	✗
RTEMS	✗	✗	✗	FreeRTOS	✓	✗	✗
Micrium µC/OS	✓	✗	✗	TI-RTOS	✓	✗	✗
DSP/BIOS	✓	✗	✗	TinyOS	✓	✗	✗
LiteOS	✓	✗	✗	RIOT	✓	✗	✗
ARM mbed	✓	✗	✗	Contiki	✗	✗	✗
Nano-RK	✗	✗	✗	Mantis	✗	✗	✗

* *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	20/42	13/42	17/42
Non-Mobile	16/36	8/36	12/36
Non-Linux/Windows/BSD	7/27	1/27	3/27
Deeply Embedded	3/20	0/20	1/20

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 de-

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

OS vs. Mitigation	Memory Protection	Virtual Memory	OS CSPRNG
All OSes	34/42	21/42	20/42
Non-Mobile	29/36	16/36	15/36
Non-Linux/Win/BSD	20/27	5/27	6/27
Deeply Embedded	13/20	0/20	1/20

pendency 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 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.

Core Family	Arch.	MPU	MMU	ESP
ARM				
ARM1	N	✗	✗	✗
ARM2	N	✗	✗	✗
ARM3	N	✗	✗	✗
ARM6	N	✗	✗	✗
ARM7	N	✗	✗	✗
ARM7T	N	~	~	✗
ARM7EJ	N	✗	✗	✗
ARM8	N	✗	✓	✗
ARM9T	N	~	~	✗
ARM9E	N	~	~	✗
ARM10E	N	✗	✓	✗
ARM11	N	~	~	✓
ARM Cortex-A	N	✗	✓	✓
ARM Cortex-R	N	✓	✗	✓
ARM Cortex-M	N	~	✗	✓
PIC				
PIC10	H	✗	✗	✗
PIC12	H	✗	✗	✗
PIC16	H	✗	✗	✗
PIC18	H	✗	✗	✗
PIC24	H	✗	✗	✗
dsPIC	H	✗	✗	✗
MIPS32				
PIC32MX	N	✗	✗	✗
PIC32MZ EC	N	✗	✓	✗
PIC32MZ EF	N	✗	✓	✓
PIC32MM	N	✗	✓	✓
PowerPC				
PPC e200	N	~	~	✓
PPC e300	N	✗	✓	✓
PPC e500	N	✗	✓	✓
PPC e600	N	✗	✓	✓
PPC 403	N	✗	✗	✗
PPC 401	N	✗	✗	✗
PPC 405	N	✗	✓	✓
PPC 440	N	✗	✓	✓
PPC 740	N	✗	✓	✓
PPC 750	N	✗	✓	✓
PPC 603	N	✗	✓	✓
PPC 604	N	✗	✓	✓
PPC 7400	N	✗	✓	✓

The *core families* in our selection belong to the following embedded architectures: ARM, MIPS32, PIC, PPC, x86,

TABLE VI: Core Family dependency support in Harvard (H) and Von Neumann (N) architectures II

Core Family	Arch.	MPU	MMU	ESP
x86				
Intel Atom Z34XX	N	×	✓	✓
Intel Quark X10XX	N	×	✓	✓
Intel Quark μC[1]	N	×	✓	✓
SuperH				
SH-1	N	×	×	×
SH-2	N	×	×	×
SH-3	N	×	✓	×
SH-4	N	×	✓	×
AVR				
ATtiny	H	×	×	×
ATmega	H	×	×	×
ATxmega	H	×	×	×
AVR32				
AVR32A	N	✓	×	×
AVR32B	N	×	✓	×
8051				
Intel MCS-51	H	×	×	×
Infineon XC88X-I	H	×	×	×
Infineon XC88X-A	H	×	×	×
m68k				
NXP M683XX	N	×	×	×
NXP ColdFire V1	N	×	×	×
NXP ColdFire V2	N	×	×	×
NXP ColdFire V3	N	×	×	×
NXP ColdFire V4	N	×	✓	×
NXP ColdFire V5	N	×	✓	×
TriCore				
Infineon TC11xx	H	×	~	×
Infineon AUDO Future	H	×	×	×
C166				
Infineon XE166	N	✓	×	✓
Infineon XC2200	N	✓	×	✓
MSP430				
MSP430x1xx	N	×	×	×
MSP430x2xx	N	×	×	×
MSP430x3xx	N	×	×	×
MSP430x4xx	N	×	×	×
MSP430x5xx	N	×	×	×
MSP430x6xx	N	×	×	×
MSP430FRxx	N	✓	×	×
Blackfin				
Analog Blackfin[2]	N	✓	×	×
ARC				
Synopsys ARC EM	H	~	×	×
Synopsys ARC 600	H	~	×	×
Synopsys ARC 700	H	×	~	×
RL78				
Renesas RL78/G1x	H	×	×	×
Renesas RL78/L1x	H	×	×	×
RX				
Renesas RX200	H	~	×	×
Renesas RX600	H	~	×	×

[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.*

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

Von Neumann Mitigation Support	Support
MPU	6/51 (11.8%)
MMU	24/51 (47.1%)
Hardware ESP	22/51 (43.1%)

(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 μC/OS 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 MMU support. A small minority of *core families* has MPU support (MPU was supported in 6 out of 51 in Von Neumann architecture), leaving just under half of the Von Neumann *core families* in our selection without necessary hardware support for memory protection and over half without the hardware support required for virtual memory. This lack of MPU and MMU support makes sense for the more constrained end of the spectrum such as MCUs, which only have support for integrated memory and no support for external memory. Similarly, under half of Von Neumann *core families* in the selection have hardware ESP support, meaning ESP can only be implemented via software emulation on these systems.

As observed by Barr [15], UBM Embedded Markets study [7] and other observers [16], the embedded world is seeing a trend towards deployment of 32-bit CPUs (and for the most high-end embedded systems even 64-bit CPUs [17]) over the traditionally used 8- or 16-bit CPUs. Since most popular 32-bit architectures are Von Neumann, this has security implications, though these are possibly offset by the fact that certain modern CPU architectures offer hardware ESP support (e.g., ARMv6+, MIPS32r3+, x86, etc.).

Based on the above observations, we can conclude that there is a gap when it comes to *deeply embedded systems*. Only among the *high-end* Linux, BSD and Windows-based OSes there are significant exploit mitigation adoption and

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

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 [18] from those in desktop or web application development. Compared to the general-purpose world, the embedded world is heavily fragmented [19] 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 [20], [21] 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 [4], 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 [21] 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 constrains 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 seen adoption 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., PaX'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 Trust-Zone*), 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. μARMOR DESIGN

In this section, we propose μArmor, an exploit mitigation and OS CSPRNG baseline design for *deeply embedded systems* in the form of LLVM passes. μArmor seeks to address the relevant gap areas based on the results from our quantitative analysis. μArmor 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, μArmor does not seek to protect against *data-flow hijacking* or *data-only* attacks.

B. High-Level Design

μArmor incorporates three mitigations measures in order to match the functionality of the baseline outlined in Section II: μESP, μScramble, and μSSP 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, μSSP assumes the presence of either an OS CSPRNG (e.g., μRNG described in Section VI-F) or a TRNG (True Random Number Generator). We provide OS CSPRNG support as part of μRNG.

The final components of μSSP 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. μRNG Design

μRNG 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, μRNG is not a simple reimplementation of Keccak [35]. μRNG 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 μRNG 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 μRNG in the context of our representative platform, the μRNG design is not restricted to any OS or platform in particular.

In μRNG 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. μRNG uses the `Keccak-f[200]` permutation with *rate* and *capacity* parameters `r = 64` and `c = 136` respectively. Note that μRNG 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 μRNG 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 μRNG is based, μRNG provides the required security against *passive* state recovery attacks as long as reseeding occurs at least every $r * 2^{\frac{r}{2}} = 64 * 2^{32} = 32GB$ of PRNG output and against *active* attacks as long as reseeding happens at least every $2^8 * r = 256 * 64 = 2KB$ 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. μRNG 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, μRNG uses non-domain specific entropy sources that can be found on most embedded devices. We can divide these sources into two groups:

a) 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, μRNG would require at least $\frac{2*128}{0.05*8} = 640$ bytes of SRAM to guarantee a security strength level of 128 bits, a reasonable restriction for most modern microcontrollers.

b) 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	Permissions
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.

TABLE IX: `TI LM3S6965` MPU μESP Settings, *execute-from-flash*

Region No.	Description	Perms.	Size
0	Default	RW + XN	4 GB
4	SCB	RO + XN	64 B
5	MPU	RO + XN	64 B
6	Code (other)	RO + X	256 KB
7	Code (sensitive)	RO + XN	*

TABLE X: `TI LM3S6965` MPU μESP Settings, *execute-from-RAM*

Region No.	Description	Perms.	Size
0	Default	RW + XN	4 GB
2	SCB	RO + XN	64 B
3	MPU	RO + XN	64 B
4	Code (other, RAM)	RO + X	* MB
5	Code (sensitive, RAM)	RO + XN	*
6	Code (other, flash)	RO + X	256 KB
7	Code (sensitive, flash)	RO + XN	*

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, μESP 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. μScramble Implementation

The μScramble 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 μSSP 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 μScramble 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. μSSP Implementation

We implemented μSSP as an augmentation of *Zephyr*'s SSP implementation. Since *Zephyr* uses the *GCC* SSP model it already meets μSSP'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. μRNG Implementation

We implemented μRNG as a driver for the *Zephyr* `random` API. μRNG output can be requested with the `sys_rand32_get` API which *squeezes* the μRNG keccak object to produce 64 bits (the *rate* minimum) of PRNG output, the upper and lower halves of which are xor-summed together to produce a 32-bit random number as per API specifications. Since our μRNG implementation uses SRAM SUVs as its initial entropy source, it is important that this entropy collection takes place as early as possible to reduce SRAM contamination (from code storing variables, using the stack, etc.) as much as possible. We chose to integrate μRNG initialization in *Zephyr*'s `__start` routine which is the firmware code entrypoint and used as the `reset handler` in the ARM Cortex-M's `vector table`. Note that almost all RTOSes provide such initialization routine. This ensures SRAM is untouched before our μRNG initialization code is invoked.

VIII. μARMOR EVALUATION & LIMITATIONS

A. Overhead Evaluation

We evaluate the overhead imposed by μArmor 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 XI. 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 XI 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[1] (A)	% CS[1] (R)	App	% CS (A)[1]	% CS[1] (R)
Application			**Test**		
lift	1.5	0	cover	1.2	0
powerwindow	2.2	0.1	duff	3	
			test3	0.4	0.1
Kernel			**Sequential**		
binarysearch	3.8	0	adpcm_dec	1.1	0
bitcount	2.3	0	adpcm_enc	1.3	0
bitonic	4.2	0	ammunition	1.1	0.1
bsort	3.5	0	anagram	1.8	0
complex_update	1.6	0	audiobeam	1.6	0
countnegative	3.4	0	cjpeg_transupp	1.2	0
fac	4	0	dijkstra	1.9	0
fft	2.6	0	epic	1.6	0
filterbank	1	0	fmref	1	0
fir2dim	1.5	0	gsm_dec	1.3	0
iir	2.2	0	h264_dec	0.8	0
insertsort	1.9	0	huff_dec	1.8	0
jfdctint	1.3	0	huff_enc	1.6	0
lms	1.4	0	mpeg2	1.1	0.1
ludcmp	0.9	0	ndes	1	0
matrix1	3	0	petrinet	0.2	0
md5	1.9	0	rijndael_dec	0.3	0
minver	0.8	0	rijndael_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			
basicmath	0.7	0			

[1] *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 [46]. *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	1.2	0	× (0 B)	0
echo_server	0.2	0	× (0 B)	0
telnet	0.2	0	× (0 B)	0
Wrt. Resources				
philosopher	0	0	0	×
echo_server	0	0	0	×
telnet	0	0	0	×

TABLE XIII: μSSP Overhead Evaluation

Wrt. Application	%Code	% Data	%Memory	%Runtime
philosopher	30.5	0	× (48 B)	0
echo_server	26.4	0	× (84 B)	0.7
telnet	27.3	0	× (84 B)	0.7
Wrt. Resources				
philosopher	0.9	0	0	×
echo_server	5	0	0	×
telnet	5	0	0	×

TABLE XIV: μRNG Overhead Evaluation

Wrt. Application	% Code	% Data	%Memory	%Runtime
philosopher	10.2	0.4	× (52 B)	0
echo_server	1.4	0.1	× (52 B)	0
telnet	1.5	0.1	× (52 B)	0
Wrt. Resources				
philosopher	0.3	0	0	×
echo_server	0.3	0	0	×
telnet	0.3	0	0	×

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 other variants the gadget still resides at the same address. We then obtained the average and maximum gadget survival rates. 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.

TABLE XV: μScramble Coverage Analysis.

Set	Avg. GS[1]	Max. GS[1]	Set	Avg. GS[1]	Max. GS[1]
Application			**Application**		
lift	2	9	cover	29.8	143
powerwindow	1.2	10	duff	1.6	7
			test3	0	0
Kernel			**Sequential**		
binarysearch	1.4	13	adpcm_dec	0.5	4
bitcount	63.3	180	adpcm_enc	0.6	5
bitonic	85.15	166	ammunition	3.6	147
bsort	58.5	171	anagram	0.6	4
complex_upda	68.1	199	audiobeam	4	79
countnegativ	2	11	cjpeg_trans	32	96
fac	2.5	4	cjpeg_wrbmp	1.4	3
fft	1.2	5	dijkstra	1.3	5
filterbank	59.9	209	epic	0	0
fir2dim	70.5	210	fmref	0.5	3
iir	2.5	8	gsm_dec	1.3	43
insertsort	101.5	202	h264_dec	100	181
jfdctint	97	193	huff_dec	18.8	91
lms	64.2	193	huff_enc	5.8	55
ludcmp	55.4	166	mpeg2	0	0
matrix1	68.8	203	ndes	0.6	2
md5	0.6	7	petrinet	2	2
minver	1.3	5	rijndael_de	2.5	16
pm	12.5	97	rijndael_en	3	11
prime	0.9	5	statemate	16.4	97
quicksort	1.5	9			
recursion	91.6	188			
sha	2.3	6			
st	0.8	3			
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 μRNG) 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 bruteforce approaches at all. The *thread restart* policy is susceptible to bruteforce 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 bruteforce 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 μRNG 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.

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.

[2] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Security Symposium*, 2014.

[3] B. Herzberg, D. Bekerman, and I. Zeifman, "Breaking down mirai: An iot ddos botnet analysis," *Incapsula Blog, Bots and DDoS, Security*, 2016.

[4] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, 2004.

[5] K. GReAT, "Equation group: Questions and answers," 2015.

[6] P. K. et al., "Deeply embedded survivability," *ARO Planning Workshop on Embedded Systems and Network Security*, 2007.

[7] U. Tech, "Embedded market study," https://goo.gl/BerqaP, 2015.

[8] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *IEEE Symposium on Security and Privacy*, 2017.

[9] V. Research, "The battle for the iot is being fought in the mcu software ecosystem," 2015. [Online]. Available: https://goo.gl/k5WiWE

[10] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, 2016.

[11] R. Yerraballi, "Real-time operating systems: An ongoing review," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000), WIP Section, Orlando Fl*, 2000.

[12] A. Elvstam and D. Nordahl, "Operating systems for resource constraint internet of things devices: An evaluation," Ph.D. dissertation, Malmö University, 2016.

[13] NetMarketShare, "Mobile/tablet operating system market share," 2017. [Online]. Available: https://goo.gl/8ldYlg

[14] S. Evanczuk, "Mcu popularity perceptions," https://goo.gl/8WLHno, year = "2013".

[15] M. Barr, "Trends in embedded software design," 2012. [Online]. Available: https://goo.gl/PsCJHB

[16] P. Clarke, "Mcu market turns to 32-bits and arm," https://goo.gl/FtJ7db, 2013.

[17] T. Simon, "Processors rule the day," 2015. [Online]. Available: https://www.semiwiki.com/forum/content/5086-processors-rule-day.html

[18] P. Koopman, "Embedded system design issues (the rest of the story)," in *Computer Design: VLSI in Computers and Processors*. IEEE, 1996.

[19] R. Van Solingen, "Integrating software engineering technologies for embedded systems development," in *International Conference on Product Focused Software Process Improvement*. Springer, 2002, pp. 466–474.

[20] R. van Solingen, "Integrating software engineering technologies for embedded systems development," *PROFES*, 2002.

[21] P. K. et al., "Embedded system design issues (the rest of the story)," *Proceedings of the 1996 International Conference on Computer Design*, 1996.

[22] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134618

[23] A. Abbasi, "Race to the bottom: embedded control systems binary security, an industrial control system protection approach," Ph.D. dissertation, Eindhoven University of Technology, 2018.

[24] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*. IEEE, 2007, pp. 169–178.

[25] W. Puffitsch and M. Schoeberl, "Time-predictable virtual memory," in *Real-Time Distributed Computing (ISORC)*. IEEE, 2016.

[26] R. D. C. et al., "Tinykey: A light-weight architecture for wireless sensor networks securing real-world applications," *WONS*, 2011.

[27] A. Francillon and C. Castelluccia, "Tinyrng: A cryptographic random number generator for wireless sensors network nodes," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops*, 2007.

[28] NIST, "Lightweight cryptography," https://goo.gl/QRoH8u, 2017.

[29] E. N. L. et al., "Scrutinizing wpa2 password generating algorithms in wireless routers," *USENIX WOOT*, 2015.

[30] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy*, 2013.

[31] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[32] T. Goodspeed, "Nifty tricks and sage advice for shellcode on embedded systems," 2013.

[33] A. Bolshev and B. Ryutin, "Practical Firmware Reversing and Exploit Development for AVR-based Embedded Devices," https://goo.gl/VV4g7z, 2015.

[34] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in *USENIX Security Symposium*, 2012.

[35] A. Van Herrewege and I. Verbauwhede, "Software only, extremely compact, keccak-based secure prng on arm cortex-m," in *51st Annual Design Automation Conference*. ACM, 2014.

[36] G. Bertoni, J. Daemen, M. Peeters, and G. Assche, "The keccak reference," *Submission to NIST (Round 3)*, 2011.

[37] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge-based pseudo-random number generators." in *CHES*, 2010.

[38] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices." in *USENIX Security Symposium*, 2012.

[39] M. Platonov, J. Hlaváč, and R. Lórencz, "Using power-up sram state of atmel atmega1284p microcontrollers as physical unclonable function for key generation and chip identification," *Information Security Journal: A Global Perspective*, 2013.

[40] A. Van Herrewege, "Lightweight puf-based key and random number generation," Ph.D. dissertation, KU Leuven, 2015.

[41] Z. Project, "Zephyr," https://www.zephyrproject.org/, 2017.

[42] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," in *Communications and Network Security (CNS), 2016 IEEE Conference on*, 2016.

[43] Zephyr, "Dining philosophers," 2017. [Online]. Available: https://www.zephyrproject.org/doc/samples/philosophers/README.html

[44] ——, "Echo server," 2017. [Online]. Available: "https://bit.ly/2PqVTzs"

[45] ——, "Sample telnet console application," 2017. [Online]. Available: https://www.zephyrproject.org/doc/samples/net/telnet/README.html

[46] H. F. et al., "Taclebench: A benchmark collection to support worst-case execution time research," *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[47] T. I. for Communication Technologies and E. Systems, "Dspstone: Dsp compiler and processor evaluation," 2017. [Online]. Available: https://goo.gl/Rk9exu

[48] M. R.-T. R. C. (MRTC), "Wcet project / benchmarks," 2017. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[49] S.-S. Lim, "Snu real-time benchmarks," 2017. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[50] M. G. et al., "Mibench," http://vhosts.eecs.umich.edu/mibench/, 2017.

[51] M. Consortium, "Mediabench," 2017. [Online]. Available: http://mathstat.slu.edu/~fritts/mediabench/

[52] M. I. o. T. Lincoln Laboratory, "High performance embedded computing (hpec) challenge benchmark suite," 2017. [Online]. Available: http://www.omgwiki.org/hpec/files/hpec-challenge/

[53] J. Salwan, "Ropgadget," 2017. [Online]. Available: https://goo.gl/BDZuRq

[54] P. Peris-López, "Avrand: A software-based defense against code reuse attacks for avr embedded devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[55] J. Habibi, A. Gupta, S. Carlsony, A. Panicker, and E. Bertino, "Mavr: Code reuse stealthy attacks and mitigation on unmanned aerial vehicles," in *Distributed Computing Systems (ICDCS)*, 2015.

[56] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon, and S. Etalle, "µshield: configurable code reuse attack mitigation for embedded systems," in *International Conference on Network and System Security*. Springer, 2017, pp. 694–709.

[57] A. Cui and S. Stolfo, "Defending embedded systems with software symbiotes," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[58] EPOXY, ""epoxy rt_edivirt_handlers.c source code"," https://goo.gl/mm73Aj, 2017.