Building Embedded Systems Like It's 1996

Ruotong Yu[†γ] Francesca Del Nin[‡] Yuchen Zhang[†] Shan Huang[†] Pallavi Kaliyar[§] Sarah Zakto[¶]
Mauro Conti[‡∗] Georgios Portokalidis[†] Jun Xu[†γ]
[†]Stevens Institute of Technology [‡]University of Padua [§]Norwegian University of Science and Technology
[¶]Cyber Independent Testing Lab [γ]University of Utah [∗]Delft University of Technology

***Abstract*—Embedded devices are ubiquitous. However, preliminary evidence shows that attack mitigations protecting our desktops/servers/phones are missing in embedded devices, posing a significant threat to embedded security. To this end, this paper presents an in-depth study on the adoption of common attack mitigations on embedded devices. Precisely, it measures the presence of standard mitigations against memory corruptions in over 10k Linux-based firmware of deployed embedded devices.**

The study reveals that embedded devices largely omit both user-space and kernel-level attack mitigations. The adoption rates on embedded devices are multiple times lower than their desktop counterparts. An equally important observation is that the situation is not improving over time. Without changing the current practices, the attack mitigations will remain missing, which may become a bigger threat in the upcoming IoT era.

Throughout follow-up analyses, we further inferred a set of factors possibly contributing to the absence of attack mitigations. The exemplary ones include massive reuse of non-protected software, lateness in upgrading outdated kernels, and restrictions imposed by automated building tools. We envision these will turn into insights towards improving the adoption of attack mitigations on embedded devices in the future.

I. Introduction

Embedded devices are running everywhere to connect the physical world with the digital world. By estimation, there may be up to 35 billion embedded devices installed in the wild [24]. This large-scale deployment makes the security of embedded devices critical to our society. Towards escalating embedded security, it is beneficial to gain a systematic understanding of the deficiencies. Past research has initiated many efforts in this direction [22], [32], [20], [17], [14], [23], [33], [21], [19], [16]. However, most of them focus on disclosing vulnerabilities in embedded devices and understanding the threats imposed by the vulnerabilities, largely ignoring the other major category of deficiencies related to the adoption of attack mitigations. This creates a gap in our understanding.

The gap was gradually realized in recent years, and attempts have been made to fill the gap. Earlier research in this line [35] brings preliminary evidence showing a lack of adoption of popular mitigations on embedded devices. More recent studies [34], [7] unveil that this lack of mitigations is tied to limited hardware or Operating System (OS) support. For instance, Abbasi et al. [7] observe that deeply embedded devices often lack hardware features such as Memory Management Unit to enable mainstream mitigations against memory corruption exploits. These works unquestionably help complete our understanding, but they (somewhat and unintentionally) leave behind an impression that the support-wise barriers are the primary blame for the absence of attack mitigations and techniques enabling mitigations without those supports (e.g., [7], [15]) can essentially solve the problem. But does this reflect the reality in general?

Aiming to investigate the above doubt, we present a large-scale study in this paper. Our angle is to look at the adoption of attack mitigations by **embedded devices with all the needed supports**, centering around three dimensions:

- *With all the needed supports available, do embedded devices adopt the attack mitigations?*
- *Is the adoption of the attack mitigations improving over time? Is the upcoming future becoming better?*
- *If the attack mitigations are observed absent, what are the possible causes*?

Design of Study: The approach of our study is to inspect firmware running on real Linux-based embedded devices, seeking to understand their adoption of the mitigations listed in Table I and Table II. Firmware is targeted to match the setup of existing studies of embedded security [35], [22], [32], [20], [17]. Linux-based devices are considered because (i) they are typically equipped with high-end hardware, which offers modern features needed by the mitigations of interest; (ii) they represent the dominant type of embedded devices, according to our data presented in §III-B. The selection of target mitigations is a choice of multiple factors. First, these mitigations, against the influential memory corruption exploits [29], [1], are standard security features in common types of computer system (e.g., desktops, servers, and mobile phones). Second, the mitigations have been integrated into standard compiling/building toolchains of Linux systems, which can be easily deployed. Third, the mitigations are released over three years ago. This ensures that the vendors have sufficient time to adopt them.

Specifically, we collect over 18k firmware images from 38 popular embedded device vendors. Unpacking the firmware images, we extract nearly 3,000k user-space binaries and 8k Linux kernels, as described in Table IV. The binaries and Linux kernels are then statically analyzed to measure the presence of attack mitigations. By breaking down the measurement results into different periods, we further gain an understanding of the evolution in the adoption of attack mitigations. Finally, we zoom into the binaries and kernels to find commonalities that can help explain the observed absence of attack mitigations.

Results and Findings: When embedded binaries are built, attack mitigations are not frequently adopted. Considering desktop binaries as the baseline, the overall adoption rates of embedded binaries are many times lower. For instance, 85.3%

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24031
www.ndss-symposium.org

TABLE I: Target attack mitigations in embedded binary programs

Attack Vector	Mitigation	First Release	Default[1]
Stack Overflow	Stack Canaries	2005 (GCC)	✓
GOT Hijacking	Relocation Read-Only	2004 (GCC)	✓
Code Injection	Non-executable Stack	2003 (GCC)	✓
Buffer Overflow	Fortify Source	2004 (GLIBC)	✓
Control-flow Hijacking	Position-Independent (or ASLR-Capable) Code	2003 (GCC)	✓

[1] Tested on Debian 10 "buster," released in July 2019, GCC v8.3.0

TABLE II: Target attack mitigations in Linux kernel

Attack Vector	Mitigation	Building Configuration	Release Version	First Release
Stack Overflow	Stack Protector	CONFIG_HAVE_CC_STACKPROTECTOR	ARM:v2.6 MIPS:v3.11 PowerPC:4.20	2009
Privilege Escalation	PXN[2]	–[1]	ARM:v3.19 AArch64:v3.7	2012
Control Flow Hijacking	KASLR	CONFIG_RANDOMIZE_BASE	ARM:v4.6 MIPS:v4.7 PowerPC:v5.2	2014
Heap Corruption	Freelist Randomization	CONFIG_SLAB_FREELIST_RANDOM	v4.7	2016
Information Leakage	USERCOPY	CONFIG_HARDENED_USERCOPY	v4.8	2016
Buffer Overflow	Fortify Source	CONFIG_Fortify_Source	AArch64&PowerPC:v4.13, ARM-32:v4.17, MIPS:v5.5	2017
Code Injection	Non-executable Memory	CONFIG_STRICT_KERNEL_RWX	ARM:v4.11 PowerPc:v4.13 (MIPS does not support)	2017

[1] "–" indicates the mitigation is not affected by the building configuration.
[2] x86/x64 have similar mitigations called SMEP and SMAP. They are not considered because no x64/x86 kernels are identified in our dataset.

of desktop binaries adopt Stack Canaries, but only 29.7% of embedded binaries do. The lack of mitigations in embedded binaries is mainly a "decision" of the device vendors, except for a few cases where the architecture and runtime offer insufficient supports. The analysis of kernels presents much worse results. The kernels rarely adopt attack mitigations. Even the most frequently applied mitigation, Stack Protector, only has an adoption rate of 5.6%. The absence of kernel-level mitigations is largely attributed to one reason. That is, the vendors broadly use older kernels where the mitigations are not available, despite newer versions supporting the mitigations already exist for a long time.

Further, our evolution analysis identifies no clear growth in the adoption of attack mitigations by embedded binaries. We hence envision their low rates of adopting attack mitigations will less likely improve in the near future. In contrast, we do observe positive changes happening to kernels. Older kernels are disappearing and newer kernels are emerging. As a result, mitigations such as Stack Protector have been applied more frequently in recent years.

Our last main finding is the following observations to help explain why vendors do not apply attack mitiations in embedded binaries.

1) Vendors of embedded devices often use automated tools to build the systems. The automated tools tend to have a huge delay in importing support of the attack mitigations. When an older version of the automated tools is used, which happens in practice, the attack mitigations are often not available and thus, cannot be adopted.
2) A large number of binaries are reused across products or even vendors. The lack of mitigations in those binaries spreads with their propagation. The vendors cannot change that unless they can rebuild the initial binaries.

Contributions: We make the following contributions.

- We present an in-depth study to measure the adoption of attack mitigations by embedded devices. The study presents a comprehensive view of the lack of attack mitigations even on platforms that support them. We believe it will help raise broader awareness of the threat behind.
- We unveil a set of key factors leading to the lack of attack mitigations. These will bring insights towards improvement and eventually benefit the security of embedded devices.
- We build an update-to-date dataset of Linux-based embedded firmware. We create a set of mitigation identification tools tailored to embedded binaries and kernels. Both the dataset and tools will be made publicly available upon publication of the paper. The dataset and intermediate results are released at https://github.com/junxzm1990/iot-security.

II. Challenges

Running our desired study has many challenges. We describe the major ones in the section.

Building a High-quality Dataset: Obtaining a high-quality dataset of firmware images is essential but complex. It requires scale, diversity, and representativeness in the firmware images. Past research [13], [17], [32] has built such datasets. However, we cannot reuse them. First, the datasets were collected years ago, which may not represent what happens at present. Second, the public datasets were released in the format of URL links to the images[1]. Most of the links are outdated and invalid today.

Unpacking Firmware Images. Unpacking the firmware images and extracting the required components is also challenging. Different vendors organize their firmware images in diverse formats, and the vendors typically do not provide information about the composition. Past efforts have made plenty of progress in addressing the challenge. Tools, such as Binwalk [28] and FIRMADYNE [13], can already unpack a broad spectrum of firmware and extract individual files like binary programs. However, they are limited in identifying and processing kernels. First, kernels in embedded firmware often have customized signatures, which existing tools cannot capture. Second, when extracted by existing tools, the kernels are usually in the form of raw data, which cannot be further parsed and analyzed.

Identifying Attack Mitigations: Binaries in embedded firmware are heterogeneous. They run different architectures (x86,

[1]Example: http://firmware.re/usenixsec14/usenixsec14-candidates.yaml.gz

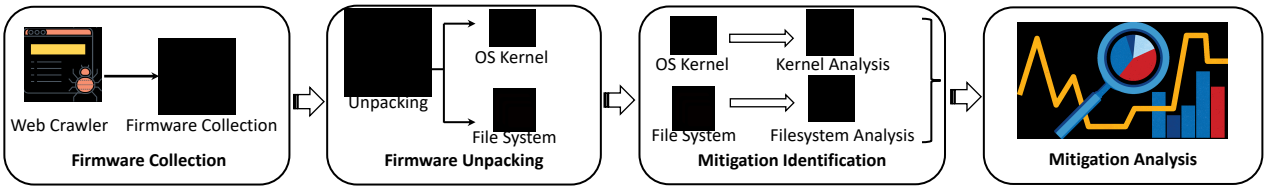


Fig. 1: Workflow of our study

ARM, MIPS, etc.) and follow various formats (stripped or not, statically or dynamically linked, using glibc or uClibc, etc.). The heterogeneities affect the identification of mitigations. For instance, the identification of Stack Canaries can be done by querying the relocation information in dynamically linked binaries, but not so for statically linked, stripped binaries. Existing tools, including Hardening-Check [36], Checksec [12], Pwntools [31], are mainly designed to work in desktop environments. They are not aware of these heterogeneities and can present reduced utilities when handling embedded binaries. Further, most existing tools do not provide presence testing of kernel-level mitigations. Checksec [12] offers such testing but requires the kernel is booted and running, which cannot scale to support a large-scale study like ours. New tools to statically identify mitigations from kernels are needed.

In the following two sections, we detail how we overcome the above challenges, following the workflow in Fig. 1.

III. Data Collection and Processing

A. Collecting Firmware Images

Our study starts with preparing web crawlers to scan the websites of mainstream embedded vendors and collect their firmware images. Such crawlers have been developed by past efforts [13]. However, due to the dynamic nature of vendors' websites, they yield poor results *today*. We update and extend the crawler released at [2] to gather firmware images from the vendors listed in Table III. We create a separate parser for each vendor website using XPath to parse a given root webpage. If the webpage contains an element matching a link to a firmware image, the parser will download the image. Concurrently, the parser will record elements about the *product name*, *firmware version*, *release time*, and *device type*, when available. Other webpages referred by this webpage will then be recursively processed in a similar way.

While downloading firmware images, we only target files with an extension of `img`, `bin`, `rar`, `pkg`, `chk`, `tar`, `zip`, `stk`, and `rmt`. Setting rules on filename extension allows us to drop obviously non-firmware content like text scripts, PDF files, and Microsoft Office documents. It also helps reduce the storage space needed to keep the downloaded data and their unpacked versions. To operate within legal and ethical boundaries, we follow the procedure presented in [17]. We only download firmware images released to the general public, and we obey the `robots.txt` directives when presented. We will release all the crawlers upon publication of this paper.

Results: In total, we collected over 18,000 firmware images from 38 vendors, as summarized in Table III. The release time of the firmware images spans two decades. The earliest image was released back in 1998 (DES-3216 by D-Link), and the most recent image just came out in 2021. Fig. 2 shows the

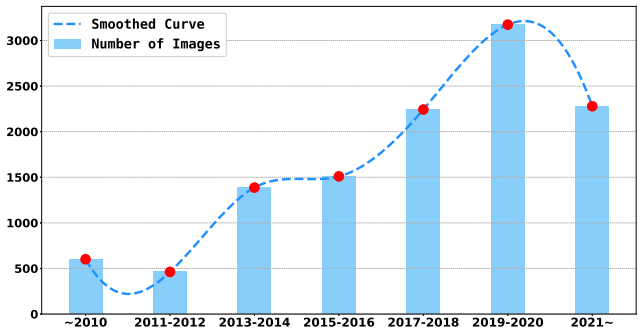


Fig. 2: Distribution of firmware images across release time. All images released before 2010 are aggregated into ~2010.

distribution of the firmware images over their release time. Clearly, more images were released in recent periods. The firmware images run on 4,000+ different products, covering many common types of device used in our daily life, as listed in Table XV in the Appendix. Among the products, over 2,000 have multiple versions of firmware available. This is important since it helps us with building an understanding of evolution over time. Overall, we envision this dataset has reasonable scale, diversity, and representativeness to support our study.

B. Unpacking Firmware Images

The next step focuses on unpacking each firmware and extracting the binary programs and Linux kernel inside. Linux-based firmware is a concatenated archive of different parts of a Linux system. As depicted in Fig. 9 in the Appendix, the archive usually includes one or more filesystem partitions, a Linux kernel, a bootloader, and various configurations and other data files. Given a firmware image, which is often compressed, we first decompress it according to the compression algorithms (`zip`, `bzip2`, `gzip`, `tar`, `rar`, etc.). Customized compression algorithms are not handled due to a lack of specifications. We then unarchive the decompressed image to extract filesystems and the Linux kernel.

Extracting Filesystems: We reuse FIRMADYNE [13], a tool built upon Binwalk [28], to extract filesystems from firmware images. Besides using manually-created signatures to locate complete filesystems, FIRMADYNE also searches for standard directories under the root directory (e.g., `bin`, `sbin`, `lib`, etc.). The filesystems and directories are then recursively traversed to identify binary programs/libraries in ELF format.

Extracting Linux Kernels: When unpacking a firmware image, FIRMADYNE can identify Linux kernels based on signatures inherited from Binwalk. However, the signatures are too specific, filtering out many kernels customized for embedded devices. We extended the signatures based on patterns observed in our dataset. Doing so enables us to identify 58.3% more Linux kernels.

TABLE III: Statistical results of firmware images collected from popular embedded device vendors. PMV means product with multiple versions of firmware available. Time Range indicates the period where the images were released (xx means the year of 20xx unless otherwise noted).

Vendor	# of Images	# of Products	# of PMVs	Time Range
Cerowrt	2	2	0	14 → 14
Haxorware	2	1	1	-
AT&T	4	3	1	-
360	5	1	1	17 → 17
Actiontec	6	5	1	-
Buffalo	6	3	2	16 → 18
camius	6	6	0	-
GOCloud	8	5	2	19 → 21
Phicomm	13	7	4	16 → 18
ZyXEL	15	7	4	17 → 21
CenturyLink	18	18	0	-
Polycom	21	4	3	18 → 19
u-blox	31	25	6	16 → 21
TENVIS	41	16	2	12 → 14
MikroTik	49	16	13	-
Foscam	83	36	22	13 → 18
AVM	107	54	44	-
RouterTech	144	15	10	06 → 11
Belkin	165	109	33	-
Linksys	166	132	38	01 → 21
Mercury	169	142	35	09 → 20
Supermicro	187	186	1	-
Digi	214	100	56	-
NETCore	255	229	21	20 → 21
Moxa	400	315	53	04 → 18
TRENDnet	409	365	46	12 → 21
Tenda	467	179	110	09 → 18
Ubiquiti	512	206	165	07 → 18
QNAP	576	27	22	16 → 18
Hikvision	607	112	77	14 → 21
Synology	672	117	99	-
TomatoShibby	692	7	7	14 → 16
Tp-Link-zh	992	566	187	08 → 23
ASUS	1099	179	146	06 → 18
D-Link	1172	260	218	98[1] → 20
Tp-Link-en	1186	274	225	-
NETGEAR	3682	663	449	-
OpenWrt	3837	73	70	20 → 21
Total	**18,020**	**4,470**	**2,174**	-

[1] "98" here means 1998.

The kernels extracted by FIRMADYNE are in raw data format, which cannot be further parsed and analyzed. To address this issue, we convert the kernels to fully analyzable ELF files with the help of Vmlinux-to-ELF [30]. At the high level, Vmlinux-to-ELF identifies symbol tables (kallsyms) in a given kernel to identify functions and then reorganizes them into an ELF file. More technical details about Vmlinux-to-ELF can be found in its manual [30].

To identify mitigations in a Linux kernel, we often need the configuration file used to build the kernel (commonly known as .config). To find the .config file, we run Binwalk to recursively extract files from the raw kernel and check whether they are .config using the file utility.

Results: We consider a firmware image unpacked if we extract any ELF files or a Linux kernel from the image. Based on this criterion, 10,685 out of 18,020 firmware images are unpacked, as reported in Table IV. The success rate is 59.3%, which is comparable to previous research (26,275 out of 32,256 images [17] and 8,893 out of 23,035 images [13]). The unpacked images are from 37 vendors and span all the major architectures (ARM, AArch64, MIPS, x86, x64, and PowerPC). For the 7,335 images that we cannot unpack, 4,277 are either non-Linux based or encrypted, which are out of our consideration. The other 3,058 contain nothing that FIRMADYNE can recognize.

From the 10,685 unpacked images, we collected 9,037 filesystems (spanning 34 vendors). The filesystems contain around 3,000k ELF binaries. More details about the binaries are presented in §V. From the unpacked images, we also extracted 7,977 Linux kernels in the format of raw data. The kernels include 99 distinct versions, ranging from v2.0.40 to v4.14.221. Vmlinux-to-ELF converted 3,287 of them to ELF files with symbols. It failed to convert the remaining kernels because kallsyms is not available (4,672 kernels) or the architectures are unknown (18 kernels). The identification of .config files is less rewarding. We only extracted .config files from 581 kernels. This is understandable since .config is typically not needed for deployment.

IV. Identification of Mitigations

A. Identifying Mitigations in User-space Binaries

Since the 2000s, the security of binary software has made leaps forward through the adoption of a variety of mitigations at the compiler and OS levels. Table I highlights the mitigations we aim to identify. We included all the mitigations that are both integrated into standard compiling/building toolchains of Linux systems and found active on modern Linux distribution. The mitigations also represent the ones concerned by the hacking communities. For instance, Pwntools [31], a popular exploit framework, considers the same set of mitigations. In the following, we introduce each of them and explain how we detect their presence.

1) Stack Canaries: Stack Canaries, also known as stack guards [18], are used to provide defense against stack overflows. This mitigation is implemented by compilers (e.g., GCC) via inserting special code at the entry and exit points of functions. At function entry, a secret, random canary value is saved at the top of the stack separating the return address from the stack frame. At function exit, the canary value is checked. If the canary value is not changed, the function returns to the caller. Otherwise, the function calls __stack_chk_fail, a routine provided by the C library, to terminate the execution and report an error.

Existing tools, including Pwntools [31] and Checksec [12], provide modules to detect Stack Canaries. They report the deployment of Stack Canaries when the symbols or relocation entries (i.e., Global Offset Table, or GOT, entries) contain __stack_chk_fail. This approach works well on binaries that are dynamically linked or non-stripped. However, many binaries running on embedded devices are statically linked and stripped. Accordingly to our analysis, existing tools completely failed to detect Stack Canaries deployed in hundreds of those binaries in our dataset.

To detect Stack Canaries in statically-linked, stripped binaries, we use a generic heuristic. We observe that, when a stack violation is detected, __stack_chk_fail prints a error message starting with "*** stack smashing detected

TABLE IV: Unpacking results. The column of `.config` shows the number of kernels with `.config` file identified. The column of `converted` shows the number of kernels that can be converted to ELF. Vendors with no image unpacked are highlighted.

Vendor	# of Images	Unpacked Images								Filesystems		Linux Kernels		
		Total	ARM	AArch64	MIPS	x86	x64	PowerPC	Other	Total	ELF (k)	Total	.config	converted
Cerowrt	2	2	0	0	2	0	0	0	0	2	0.4	0	0	0
Haxorware	2	1	1	0	0	0	0	0	0	1	0.2	0	0	0
AT&T	4	4	0	0	4	0	0	0	0	4	0.6	0	0	0
360	5	4	0	0	4	0	0	0	0	4	0.5	4	0	2
Actiontec	6	5	2	0	3	0	0	0	0	5	0.4	0	0	0
Buffalo	6	4	0	0	4	0	0	0	0	4	0.5	4	0	2
Camius	6	6	0	0	6	0	0	0	0	6	0.5	6	0	6
GOCloud	8	7	0	0	0	7	0	0	0	7	0.9	0	0	0
Phicomm	13	8	2	0	6	0	0	0	0	8	1.9	3	1	3
ZyXEL	15	8	8	0	0	0	0	0	0	7	0.8	7	0	3
CenturyLink	18	7	0	0	7	0	0	0	0	7	0.8	2	0	1
Polycom	21	16	0	0	0	0	0	0	16	0	0	16	16	0
u-blox	31	0	0	0	0	0	0	0	0	0	0	0	0	0
TENVIS	41	31	27	0	4	0	0	0	0	25	0.9	31	0	0
MikroTik	49	32	8	0	12	4	0	4	4	32	4.3	0	0	0
Foscam	83	10	10	0	0	0	0	0	0	0	0	10	0	0
AVM	107	22	15	0	7	0	0	0	0	22	5.0	0	0	0
RouterTech	144	143	0	0	143	0	0	0	0	143	25.8	142	0	0
Belkin	165	67	6	1	60	0	0	0	0	60	7.9	60	0	33
Linksys	166	115	58	0	57	0	0	0	0	74	17.1	101	24	75
Mercury	169	27	0	0	27	0	0	0	0	27	1.5	27	0	27
Supermicro	187	187	185	0	0	0	0	0	2	5	1.3	187	7	9
Digi	214	3	0	0	3	0	0	0	0	3	1.5	5	1	2
NETCore	255	153	1	0	152	0	0	0	0	152	10.2	138	1	85
Moxa	400	107	83	0	20	0	0	0	4	107	32.0	0	0	0
TRENDnet	409	169	38	0	130	0	0	0	1	142	15.3	158	3	70
Tenda	467	252	64	0	188	0	0	0	0	252	33.6	142	0	118
Ubiquiti	512	479	137	0	342	0	0	0	0	479	204.7	449	59	436
QNAP	576	297	0	0	0	297	0	0	0	297	296	0	0	0
Hikvision	607	190	189	1	0	0	0	0	0	0	0	190	41	186
Synology	672	671	346	24	0	15	250	36	0	671	1375.4	0	0	0
TomatoShibby	692	692	118	0	574	0	0	0	0	692	127.8	314	0	23
Tp-Link-zh	992	494	175	0	319	0	0	0	0	464	65.7	385	53	325
ASUS	1,099	1,069	388	0	678	0	0	0	3	1,069	273.2	438	54	288
D-Link	1,172	134	52	0	68	0	0	14	0	86	15.9	116	11	92
Tp-Link-en	1,186	660	107	3	548	0	0	2	0	654	76.3	565	43	544
NETGEAR	3,682	1,474	443	12	932	5	31	51	0	980	173.9	1,293	269	957
OpenWrt	3,837	3,335	276	18	3,021	0	0	20	0	2,546	191.2	3,184	0	0
Total	**18,020**	**10,685**	**2,540**	**58**	**7,321**	**328**	**281**	**127**	**27**	**9,037**	**2,964**	**7,977**	**581**	**3,287**

***". The message is not influenced by optimization level, CPU architecture, or the type of binary. We, thus, search for that string in a given binary. Once found, we disassemble the binary to locate the function (i.e., `__stack_chk_fail`) that uses the string. If the function is identified and called by other functions, we consider Stack Canaries are deployed. Listing 1 in the Appendix shows an example of `__stack_chk_fail` in a statically-linked ARM32 binary, which shows a pointer to the above string is used.

2) Relocation Read-Only: Relocation Read-Only (RELRO) is a defense measurement against GOT hijacking [10], applied when linking binaries. GOT holds the addresses of variables and functions that are unknown during linking but relocated at run-time (e.g., variables and functions imported from libraries). In contemporary systems, the GOT often splits into two sections: `.got` and `.got.plt`, with the latter being used by the Procedure Linkage Table (PLT). Briefly, the PLT includes code that enables *lazy binding* of external functions. Specifically, when an external function is called for the first time, code in the PLT executes and calls the linker for resolving the symbol and writing the function's address to the GOT entry (hence, *lazy*). Next time the external function is called, the PLT code directly jumps to the address in the GOT entry. Because symbols are resolved at run time, the GOT needs to be writable, making it vulnerable to attacks that corrupt the GOT entries to hijack program execution.

RELRO [3] aims to turn parts or all of the GOT read only (RO) to protect the function pointers from overwrites. This is done by resolving symbols within the GOT at load time and remapping it to RO before the program executes. There are two versions of RELRO: *partial* and *full* RELRO. Partial RELRO only protects the `.got` section, which stores offsets to symbols of variables, leaving `.got.plt` writable to perform lazy binding. Full RELRO keeps a single `.got` section and protects all of it, requiring that all symbols are resolved in the beginning. Fig. 10 in the Appendix shows the difference between no RELRO, partial RELRO, and full RELRO.

RELRO is implemented by the linker based on the meta-data found in ELF binaries, which specifies the GOT to be mapped with the designated protections (e.g., RO). We observe that for RELRO to be present, `.got` needs to be mapped to a RO segment (`GNU_RELRO`), while the concurrent

presence of a `.got.plt` in a writable segment indicates that we only have partial RELRO. Additionally, binaries with full RELRO require external symbols to be resolved at load time, which is enabled by one of the following linking flags being present in the `.dynamic` section: `BIND_NOW`, `DT_BIND_NOW`, `DF_1_NOW`. Our study relies on the appearance of `.got` and .got.plt, the protections of their segments, and the linking flags, as specified above, to identify RELRO.

3) Non-executable Stack: Most modern processors [7], including microprocessors of the ARM-Cortex-R and ARM-Cortex-M families, several MIPS32, and most PowerPC processors, support data-execution prevention (DEP) [9]. DEP is a feature that prevents the execution of instructions from protected memory segments, in particular, segments that are also writable.

The use of DEP for the program stack is most crucial to defang stack overflow vulnerabilities, commonly known as Non-executable (NX) Stack [4]. To enable NX Stack, binaries need to specify that the stack is NX explicitly. To safely detect the adoption of NX stack, we need to confirm DEP support from the hardware and the presence of `PT_GNU_STACK` in the program header for the stack segment (which mandates the stack is NX). However, due to the lack of hardware specifications, our study only checks the program header. This should not cause many problems since DEP is a pretty standard feature on processors that can run Linux-based systems.

4) Fortify Source: The standard C library (libc) includes many unsafe functions that can lead to overflows when misused. Fortify Source [5] is a defense measurement activated by compilers like GCC to check on known unsafe functions in libc. These mainly include functions that copy or write data to a destination buffer without limiting the number of bytes (e.g., `strcpy`, `strcat`, `memcpy`, etc.). Fortify Source replaces those functions with safer versions that perform size checks. Fig. 11 in the Appendix depicts one such example, where two functions are replaced with their safer counterparts.

Existing tools detect Fortify Source based on symbols or relocation entries of replacement functions in the form of `__*_chk`. Similar to the identification of Stack Canaries, this approach is effective with dynamically linked or non-stripped binaries because `__*_chk` are defined in libc and their symbols are imported. However, it cannot handle statically-linked, stripped binaries. To this end, we again apply the heuristic we used to detect Stack Canaries. The replacement functions output a constant message "`*** buffer overflow detected ***`", when detecting violations. We follow this message as an indicator to locate `__*_chk` functions and, in turn, identify the adoption of Fortify Source.

5) Position-Independent (or ASLR-Capable) Code: Address Space Layout Randomization (ASLR) [26] is a seminal defense for mitigating exploitation. ASLR mandates that each time a program executes, the code segment, the stack, the heap, and the libraries are located at a randomly selected offset in memory. Besides OS support, ASLR requires binaries (programs or libraries) to be compiled as Position Independent Executable (PIE) and Position Independent Code (PIC), or otherwise, *relocatable* code. Whether a binary is position-independent or relocatable is indicated by its program header. Specifically, position-independent or relocatable binaries have type `ET_DYN`. Otherwise, they will have type `EX_EXEC`. This enables us to identify ASLR-capable binaries by checking their program headers.

TABLE V: Adoption rates of user-space mitigations (%). The best result for each mitigation is highlighted. **Ave (Vendor)** shows results averaged on vendors while **Ave (Binary)** indicates results calculated on all binaries.

Vendors	ELF (k)	Canary	RELRO	NX	Fortify	PIE
Haxorware	0.2	0	0	0	0	14.9
Actiontec	0.4	0.5	0	47.2	0.5	13.4
Cerowrt	0.4	0	0	0	0	9.8
360	0.5	60.0	0	0	0	8.9
Buffalo	0.5	0	0	45.8	0	6.0
Camius	0.5	11.9	0	92.1	1.3	11.9
AT&T	0.6	0	0	0	0	6.3
CenturyLink	0.8	0	0	0	0	0.6
Zyxel	0.8	1.0	0	97.3	0.9	11.6
GOCloud	0.9	0	0	98.2	0	14.9
TENVIS	0.9	0	0	0	0	34
Supermirco	1.3	19.4	3.2	97.8	16.1	18.5
Digi	1.5	0	0	3.5	0	18.5
Mercury	1.5	0	0	0	0	31.5
Phicomm	1.9	0.1	0.8	21.2	0	47.2
MikroTik	4.3	0.2	7.9	81.0	0.07	5.8
AVM	5.0	81.5	89.4	95.6	0.04	90.8
Belkin	7.9	0.2	3.8	7.4	1.6	11.0
NETCore	10.2	11.3	0.02	0.06	0.2	16.4
TRENDnet	15.3	0.4	0.3	10.1	0.05	13.6
Dlink	15.9	0.4	0.4	30.4	0.04	9.1
Linksys	17.1	0.5	3	60.4	0.8	9.0
RouterTech	25.8	0	0	0	0	15.0
Moxa	32.0	39.3	15.0	75.7	35.5	31.8
Tenda	33.6	0.6	2.3	30.5	0.01	11.7
Tp-Link-zh	65.7	2.9	0.4	38.7	0.1	18.3
Tp-Link-en	76.3	0.5	0.9	36.6	0.6	21.5
TomatoShibby	127.8	0.1	1.0	23.2	0	8.4
NETGEAR	173.9	2.2	4.4	55.9	0.5	11.4
OpenWrt	191.2	0	0	99.9	0	0
Ubiquiti	204.7	6.7	1.0	15.6	25.0	9.5
ASUS	273.2	1.3	1.4	46.8	0.05	8.3
QNAP	296.0	80.1	3.1	99.2	1.4	7.7
Synology	1375.4	43.6	36.7	99.5	43.5	13.5
Ave (Vendor)	87.2	10.7	5.2	41.5	3.5	16.5
Ave (Binary)	-	29.7	18.3	76.2	22.5	11.6
Debian	34.0	85.3	98.1	99.7	55.6	94.0

B. Identifying Mitigations in the Kernel

Linux kernel has been gradually incorporating various attack mitigations since version 2.6. We examined all popular kernel mitigations [6] and targeted those (i) applicable to deployed systems (ii) active in modern Linux distributions and (iii) released over three years ago (such that the vendors have sufficient time to deploy them). Table II summarizes the ones we finally picked. Identification of these kernel-level mitigations can be done more systematically. As per Table II, the presence of the mitigations can be identified based on the architecture, the kernel version, and the building configurations. Since Vmlinux-to-ELF already provides the architecture information, we do not have to worry about it. In the following, we describe the recovery of kernel version and the identification of mitigations with and without the configuration files.

TABLE VI: Adoption rates of user-space mitigations by different types of binary (%). `Exe` and `Lib` stand for executable and library, respectively; "-" indicates the mitigation is not applicable or not meaningful.

	Type	ELF (k)	Canary	RELRO	NX	Fortify	PIE
Emd.	Dynamic Exe	1,340.3	30.9	15.4	75.5	26.1	11.7
	Dynamic Lib	1,615.0	28.8	20.7	-	19.6	-
	Static Exe	7.9	8.8	3.9	42.4	8.4	0
Debian	Dynamic Exe	20.0	89.9	98.7	99.9	75.2	94.0
	Dynamic Lib	14.0	79.8	98.1	-	30.2	-
	Static Exe	0.2	24.1	38.8	87.1	24.6	0

1) Recovery of Kernel Version: When `.config` is recovered, the kernel version is explicitly documented within. However, as we pointed out before, the `.config` file is not always available. When the `.config` is missing, we instead search for string constants within the kernel image to infer its version. For instance, kernels frequently include string resembling *Linux version 2.6.36 (root@automake) (gcc version 4.6.3 (Buildroot 2012.11.1)) #2 Fri Jan 20 15:50:29 CST 2017*, which gives explicit information about the kernel version.

2) Analysis with Building Configuration: An option in the `.config` file being selected (e.g., `CONFIG_HAVE_CC_STACK_PROTECTOR=y`) means the corresponding feature is enabled. In contrast, an un-selected option, indicated by its appearance in a line starting with "#" or its absence in the file, means the feature is not enabled. With the support of the `.config` file, we can easily determine whether a target mitigation is activated in the kernel by checking the associate options specified in Table II.

3) Analysis without Building Configuration: When the `.config` file is missing, we may still measure the presence of many mitigations based on the ELF file converted from the kernel. Consider Stack Protector as an example. We can detect its presence by the existence and usage of indicator function `__stack_chk_fail`. Similarly, Fortify Source, Vmap Kernel Stack, USERCOPY, Heap Freelist Obfuscation, Executable Memory Protection, and KASLR can be respectively detected using `**_chk`, `free_vm_stack_cache`, `usercopy_warn`, `freelist_state_initialize`, `mark_rodata_ro`, and `rotate_xor` as indicator functions. Finding indicator functions in the converted ELF is straightforward since Vmlinux-to-ELF already recovered the symbols.

V. Measuring Adoption of User-Space Mitigations

We run the identification of user-space mitigations on the 3,000k binaries described in §III. Binaries released before a mitigation are excluded from the analysis of that mitigation. For NX Stack and PIE, only executables are considered because the two mitigations are less meaningful for libraries. For other mitigations, all binaries, including both executables and libraries, are considered. In addition, full RELRO and partial RELRO are aggregated together. To build a baseline for references, we further run the identification on 34k ELF binaries extracted from 7,483 Debian packages located in the stable distribution for desktop. The Debian binaries mostly run on x86/x64 architectures. The measurement results are summarized in Table V.

A. Analysis of Results

Overall, the adoption rates of attack mitigations by embedded binaries are not high, significantly falling behind the desktop binaries (represented by Debian). Stack Canaries, one of the most common mitigations in desktop binaries, are only applied to 29.7% of the embedded binaries. Zooming into the results, even this 29.7% adoption rate is likely an overestimation of the general reality as it is exaggerated by the large number of binaries with Stack Canaries from QNAP and Synology. Without counting the two vendors, the adoption rate drops to 3.28%. The situation of RELRO and PIE is similar. The adoption rates are 98.1% and 94.0% on desktop binaries, dramatically dropping to 18.3% and 11.6% on embedded binaries.

Regarding the adoption of Fortify Source, embedded binaries also fall behind desktop binaries but to a less significant extent (22.5% v.s. 55.6%). A factor contributing to the gap is the broad use of uClibc by embedded binaries (about 22%). Unlike Glibc, uClibc does not support Fortify Source. Without considering binaries that use uClibc, the adoption rate of Fortify Source raises to 36.5%. The results of NX Stack are somewhat surprising. NX Stack, a straightforward, no-cost mitigation, is missing on about 24% of the embedded binaries. According to the breakdown results shown in Fig. 3, the lack of NX stack mostly happens to MIPS binaries. This is attributed to a MIPS-specific hardware restriction. The MIPS standards do not mandate the behaviors of Floating Point Unit (FPU) instructions. To normalize the behaviors of FPU operations, the Linux system emulates certain FPU instructions and places the emulated code on the stack to execute [34]. Thus, the stack is marked executable, and the situation only changes after a patch in Linux kernel became available in 2016.

The above analysis shows a picture from the high level. To gain a deeper understanding, we break down the results from the dimensions of binary type, architecture, and vendor.

1) Breakdown by Binary Type: Table VI shows the results separately measured on different types of binary. The majority of embedded binaries are dynamic libraries and dynamically linked executables (for simplicity, we call them dynamic executables), which adopt attack mitigations more often than their static counterparts. In particular, their adoption rates of Stack Canaries and RELRO are significantly higher. However, regardless of which type we consider, embedded binaries consistently have a lower adoption rate than desktop binaries on every mitigation.

2) Breakdown by Architecture: Embedded devices use very diverse architectures, which affects the adoption of mitigations in different ways. For instance, as pointed out above, MIPS restricts the deployment of NX Stack. Inspired by this, we break down the measurement results based on architectures.

As shown in Fig. 3, embedded binaries running on every architecture have a relatively low rate of adopting attack mitigations (except NX Stack). However, the adoption rates do vary across architectures. ARM binaries constitute the largest group. They have a moderate level of adoption rate regardless of which mitigation we consider. MIPS binaries are the second largest group. They, however, have the lowest adoption rate in nearly every mitigation. This is not surprising regarding NX Stack since MIPS imposes some hardware restrictions. The low

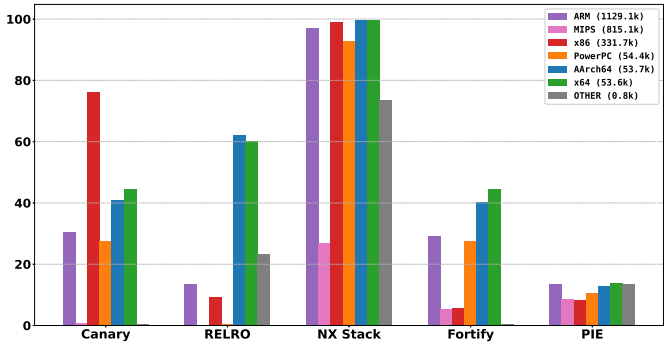


Fig. 3: Adoption rates of user-space mitigations by binaries running on different architectures. The numbers in the legend represent the number of binaries with each architecture.

adoption rates of other mitigations, in contrast, should reflect the choice of the vendors.

Compared to MIPS binaries, x86 binaries have broader adoption of the mitigations. In particular, x86 binaries have the highest adoption rate of Stack Canaries among all the architectures. However, the observation may not reflect the general case. Most of the x86 binaries come from QNAP (see Table IV), which offers an 80.1% adoption rate of Stack Canaries (see Table V). It is unclear whether the adoption rate will remain high when more x86 binaries from other vendors are considered[2].

AArch64 binaries and x64 binaries have relatively higher adoption rates in all the mitigations. In fact, they present the highest adoption rates on RELRO, NX Stack, Fortify Source, and PIE. This is reasonable since the two architectures were released more recently. Binaries running on them tend to have newer building environments (e.g., compiler and linker) and newer execution environments (e.g., libraries and OS), where the mitigations are better ready.

3) Breakdown by Vendors: Another interesting angle is to look at the differences across vendors, which helps answer questions such as which vendor offers the best mitigations. Related results have been presented in Table V.

The adoption rates of mitigations dramatically vary across vendors. Vendors like AVM and Synology apply many attack mitigations to most of their binaries. Others vendors like RouterTech and TomatoShibby rarely adopt most mitigations. Zooming into individual mitigations, the difference is similarly intense. Consider Stack Canaries and Fortify Source as examples. AVM provides Stack Canaries to 81.5% of its binaries, while 11 other vendors entirely omit Stack Canaries; Synology enables Fortify Source for 43.5% of its binaries, but in contrast, 27 other vendors only apply Fortify Source to less than 1% of their binaries. These differences reflect that the low adoptions rates of many vendors are a result of their (intentional or unintentional) "choices" instead of objective constraints.

Different vendors also have diverse "preferences". For instance, AVM prevalently applies Stack Canaries and RELRO while largely neglecting Fortify Source. On the contrary, Moxa and Synology prioritize Fortify Source but emphasize less on the AVM's preferred mitigations. Another generic observation is that more vendors have preferences for NX Stack and PIE. Presumably, this is because NX Stack and PIE have a lower cost, which is more amendable to embedded devices.

[2]We attempted to expand the dataset of x86 binaries, but could not identify many other vendors using x86 architectures.

TABLE VII: Adoption of mitigations by different device types.

Device Type	Canary	RELRO	NX	Fortify	PIE
Routers	4.1	8.0	58.5	0.1	11.0
WIFI Systems	0.4	1.4	35.7	0.2	11.4
Net-Switches	8.0	10.0	77.6	2.6	25.8
Modems	0	0	83.2	0	3.4
Net-Controllers	3.5	2.0	11.4	2.0	30.3
Less-Networked	10.4	9.9	48.8	0.1	20.6

4) Breakdown by Device Types: The adoption of attack mitigations may also be tied to the use scenarios of the devices. For instance, less-networked devices such as radio players have a lower risk of exploitation and thus, may skip the mitigations. To this end, we separately measured the attack mitigations on different types of devices. As summarized in Table VII, the adoption rates of attack mitigations are not evidently disparate across device types. All types of devices present an insufficient adoption of Stack Canary (0-10.4%), RELRO (0-10%), and Fortify Source (0-2.4%). Network switches and modems demonstrate a relatively higher adoption of NX Stack. This is not because of their types but, instead, that network switches and modems are more ARM-based than MIPS-based. Moreover and very interestingly, the adoption rates by internet-exposed devices are not higher than their less-networked counterparts. In summary, the results show no strong correlation between mitigation adoption and device type.

5) Mitigation Adoption and Vulnerability Presence: The presence of vulnerabilities is the key motivation of mitigations. This brings up two questions. First, *are the low adoption rates of mitigations attributed to the lack the vulnerabilities*? Second, *are the adoption rates higher on devices containing more vulnerabilities?* Accordingly, we perform a case study on embedded vulnerabilities. It shows that memory corruption vulnerabilities are common on embedded devices. For instance, two recently reported cases [29], [1] both appear in millions of embedded devices. This brings a negative answer to the first question. In a follow-up step, we identified 1,360 binaries packaged as part of the Realtek SDK from 369 devices, which are affected by one of the above vulnerabilities [29]. Compared to other binaries, these binaries present no broader adoption of the attack mitigations (Stack Canaries: 3.50%; RELRO: 1.60%; NX Stack: 24.40%; Fortified Source: 0.10%; PIE: 2.50%). This indicates a negative answer to the second question.

Summary: Embedded devices have a low rate of adopting user-space mitigations, despite these devices broadly have the needed supports. The low adoption rate is partially attributed to the restrictions of architectures and runtime environments, but it in general reflects the "decisions" of vendors.

B. Changes over Time

While the overall rates of adopting user-space mitigations are not exciting, the situation might be improving. To verify this, we run a separate study to inspect the evolution over time.

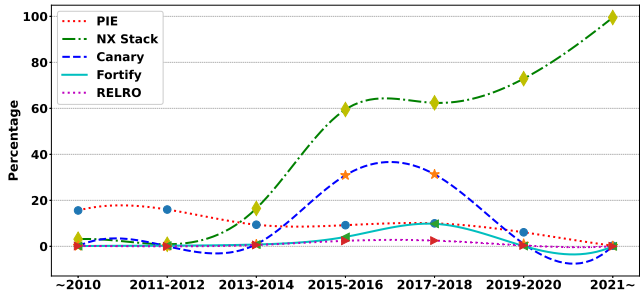


Fig. 4: Adoption rates of user-space mitigations across time. All binaries released before 2010 are aggregated into ∼2010.

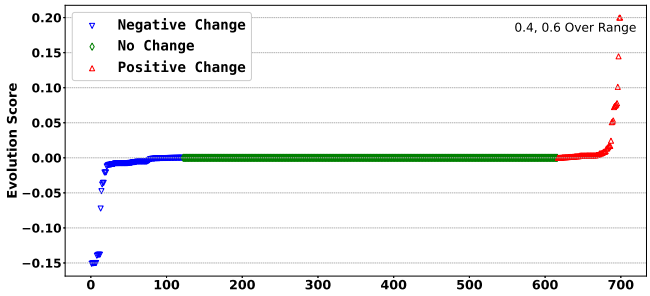


Fig. 5: Evolution score of individual firmware in the adoption of Stack Canaries. Each point represents a firmware with multiple versions. The firmware is sorted based on the evolution score. The two points marked `Over Range` at the upper left corner have evolution scores of `0.4` and `0.6`.

1) The Overall Trend: In this analysis, all binaries are grouped based on the releasing time of their firmware[3]. Specifically, binaries released in each period of two years are grouped together. Each group is then separately measured to understand their adoption of different mitigations.

Fig. 4 shows the changes over time. The adoption of NX Stack presents a consistent, positive trend. The adoption rate increased from nearly 0% before 2010 to almost 100% recently. We believe a major reason is the increasing use of newer Linux kernels (see Fig. 7). The new kernels bring better supports for NX Stack, particularly a patch to enable NX Stack in MIPS binaries [11].

The adoption rates of Stack Canaries and Fortify Source have a jump between 2015 and 2018. However, the jump may not represent what happens in general. From 2016 to 2018, QNAP released a large number of binaries with broad adoption of Stack Canaries (see Table IV and Table V), pulling up the average adoption rate. In a similar way, Ubiquiti and Moxa raised the adoption rate of Fortify Source between 2016 and 2018. During 2018-2021 where these vendors released fewer binaries, the adoption rates of both mitigations dropped. RELRO has a stable, low adoption rate in the past decade. In contrast, PIE is more often adopted, but it presents a decreasing trend. Again, this decreasing trend may overfit the decisions of certain vendors. For instance, OpenWrt released 190k binaries without PIE between 2020 and 2021, resulting in the lowest adoption rate in the past decade.

[3]Firmware with unknown releasing time is excluded.

TABLE VIII: Evolution of individual binaries in adopting attack mitigations. `No Change`, `Positive Change`, and `Negative Change` show the number of binaries without changes, with mitigation added, and with mitigation dropped.

Category	Canary	RELRO	NX	Fortify	PIE
`No Change`	278,877	278,992	278,375	279,434	278,674
`Positive Change`	438	323	1,006	61	810
`Negative Change`	284	284	218	104	115

As described above, the overall trend can be significantly affected by the decisions of specific vendors at certain points and thus, may not show the actual evolution. To this end, we perform two more fine-grained analyses.

2) Evolution of Individual Firmware: In our dataset, 699 firmware has multiple versions. The changes in mitigation adoption across different versions are good indicators of evolution. For each of the 699 firmware, we measure the *evolution score*, namely the increase/decrease of adoption rate from its earliest version to its latest version. Fig. 5 shows the results of evolution score for Stack Canaries, and Fig. 12 in the Appendix presents the results for other mitigations.

Most of the firmware presents no changes in adopting user-space mitigations. Among the few that indeed show changes, we observe both positive and negative trends. Consider Stack Canaries as an example. 83 of the firmware has an increased adoption rate, while 121 has a reduced adoption rate. Breaking down the results to individual vendors (see Table XIV in the Appendix), we observe that only Moxa offers a consistent, meaningful increase of adoption rate for all mitigations. Most vendors either do not change or change positively for some mitigations but negatively for the others. Vendors like QNAP even consistently reduce the adoption rates of mitigations when upgrading their firmware.

3) Evolution of Individual Binaries: The same binary may propagate across different versions of the firmware, which we call *versioned binaries*. The change in mitigation adoption by versioned binaries is another indicator of evolution. From our dataset, 279k versioned binaries are identified[4] from 24 vendors. Each binary is then measured to see the change of mitigation adoption between its first and last versions. Table VIII shows the aggregated results. Over 99.9% of the versioned binaries present no changes in adopting the attack mitigations. Among the remaining, a significant portion shows negative changes. More details for each vendor are presented in Table XVIII in the Appendix. Overall, no single vendor brings significantly broader adoption of mitigations to those binaries during upgrading.

Summary: There is no obvious evidence showing that the adoption of user-space mitigations is improving.

[4]Binaries with the same name are considered as the same binary.

TABLE IX: Adoption of kernel-level mitigations. `Analyzed` indicates how many kernels can be analyzed to test the presence of each mitigation; `Unsupported` means how many kernels have a version before the mitigation is integrated; `Protected` shows how many kernels adopt each mitigation.

Category	Total	Stack Protector	PXN	KASLR	FreeList	Usercopy	Fortify	Kernel RWX
`Analyzed`	3,347	2,831	839	2,062	2,063	1,980	525	564
`Unsupported`	-	2,078	798	2,048	2,049	1,968	521	555
`Protected`	-	159	41	0	0	3	4	9

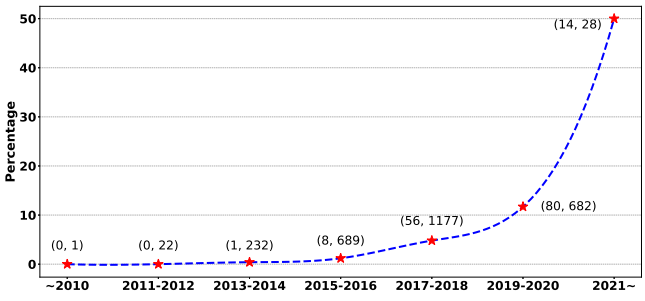


Fig. 6: Evolution of Stack Protector across time. The pair of value `(x, y)` above each point means the total number of applicable kernels is `x` and `y` of them are protected. The leap in 2021 may not reflect the general case since the total number of kernels is too small.

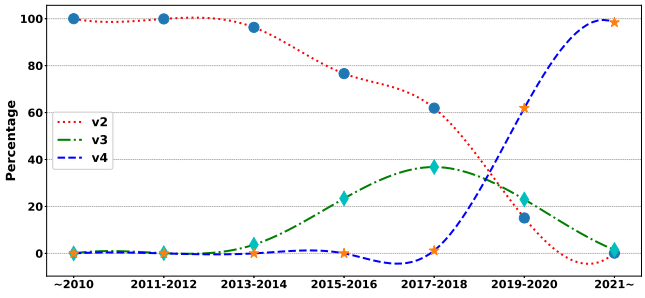


Fig. 7: Distribution of kernel versions across their building time. Subversions are disregarded and aggregated (e.g., all v2.x.y are aggregated into v2). The leap of v4 between 2020 and 2021 is presumably attributable to a large number of Linux v4 firmware released by OpenWrt.

VI. Measuring Adoption of Kernel-Level Mitigations

From the collected firmware images, 7,977 Linux kernels are extracted. We run an analysis on the kernels to identify the mitigations in Table II. We only consider kernels that have `.config` files extracted or can be converted to an ELF file as our identification approach relies on that. Given a target mitigation, we only include kernels built after the release of the mitigation and only include kernels using the desired architecture. Otherwise, the mitigation is certainly not adopted.

A. Analysis of Results

Table IX presents the results. Kernel-level mitigations are rarely adopted in embedded devices. Stack Protector is applied the most, but sill only to 159 out of 2,831 kernels. The other mitigations have an adoption rate close to zero. In particular, KASLR and Freelist Randomization are not adopted at all.

TABLE X: Gap between the release time and the building time of kernels from our dataset (months). **Gap Range** shows the range of the gaps for all kernels from the same vendor.

Vendor	# of Kernels	Gap Average	Gap Range
CenturyLink	1	81.0	[81, 81]
Phicomm	3	56.7	[32, 70]
360	4	69.0	[46, 78]
Buffalo	4	46.5	[23, 86]
Digi	5	80.5	[67, 102]
Camius	6	53.2	[36, 64]
Zyxel	7	79.4	[15, 101]
Polycom	16	115.5	[69, 185]
TENVIS	27	79.4	[71, 87]
Mercury	27	49.6	[20, 64]
Belkin	57	63.1	[15, 206]
Linksys	97	91.6	[27, 187]
Dlink	114	71.1	[18, 153]
Netcore	137	84.1	[49, 168]
RouterTech	142	102.4	[68, 132]
Tenda	142	81.5	[28, 121]
TRENDnet	147	90.1	[4, 177]
Hikvision	176	61.9	[20, 109]
Supermirco	187	110.7	[26, 174]
TomatoShibby	276	83.9	[42, 93]
Tp-Link-zh	385	71.7	[29, 138]
ASUS	415	78.9	[29, 133]
Ubiquiti	426	77.2	[16, 174]
Tp-Link-en	565	82.2	[29, 139]
Netgear	1,263	68.0	[12, 166]
Openwrt	2,755	40.2	[36, 47]
Average	284	65.1	-

A major reason why kernel-level mitigations are missing is the vendors' tendency to use old kernels. Fig. 14 in the Appendix shows the distribution of kernels across versions. Nearly half of the kernels have a version of v2.x, which was released almost 20 years ago. Looking closely at the kernels, we further find that they were mostly built years after the release, as illustrated in Table X. The average gap between the kernels' release time and building time is over 5 years (65.1 months). The gap for some kernels from Polycom and Linksys even goes over 15 years.

A consequence of using old kernels is that the mitigations are unsupported. As shown in Table IX, most kernels miss mitigations because they are not new enough to have the mitigations. A very possible incentive for the vendors to use

old kernels is reliability. When an older kernel runs well on the products, it is often safer to continue using it since an upgrade can easily introduce backward-incompatibility issues. To verify this intuition a bit, we again check the firmware with multiple versions but focus on the kernels this time. As shown in Fig. 15 in the Appendix, the vendors are in general "reluctant" to use newer kernels when upgrading their firmware, indirectly supporting our intuition.

B. Changes over Time

To understand the evolution of kernel-level mitigations, we perform an additional time series analysis. We only consider Stack Protector in this analysis because other mitigations have too few samples. The results, presented in Fig. 6, show a positive trend. The adoption rate of Stack Protector consistently increases over the past decade. The driving force behind the trend is mainly the upgrading of kernels. As demonstrated in Fig. 7, vendors are using more new kernels where Stack Protector is more prevalently integrated. We envision this trend, in the longer term, will also benefit other mitigations.

Summary: Kernel-level mitigations are missing in embedded devices. A major reason is the vendors largely use old kernels where the mitigations are not ready. Nonetheless, preliminary evidence shows positive changes are happening.

VII. Discussion: Why Mitigations are Missing

The key takeaway of our study is that attack mitigations are prevalently missing on embedded devices. Based on what we have explained, the lack of kernel-level mitigations is primarily caused by the excessive use of old kernels. In contrast, the problem of lacking user-space mitigations is more complicated. Hardware/runtime restrictions (e.g., the restriction by MIPS on NX Stack) are undeniably a factor, but the primary reason should be the "decision" of vendors. The key question worth discussing here is *why the vendors make such a decision*?

Without comments from the vendors[5], it is hard to get the exact answer to the above question. But throughout analysis of the commonalities shared by the firmware, we are able to gain some observations that may help answer the question.

A. Restrictions of Building Tools

Vendors often rely on automated tools to build their embedded systems. Buildroot [25] is one of the most popular tools for Linux-based embedded systems. These automated tools may delay the availability of attack mitigations for years. Consider Buildroot as an example. As shown in Table XVI in the Appendix, it does not offer full support of Stack Canaries until 2013 (8 years after the release of Stack Canaries). This similarly happens to other mitigations. In this regard, the use of automated tools like Buildroot (in particular the older versions) defers or even prevents the adoption of attack mitigations.

To gain a more concrete understanding, we zoom into the firmware in our dataset generated with Buildroot. In total, there are 690 of them. As shown in Fig. 8, most firmware was built with Buildroot released at or before 2012, when no mitigation was supported. In result, all binaries built together with the firmware would carry zero attack mitigations.

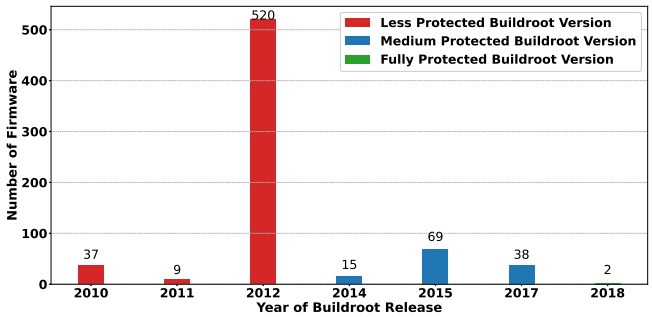


Fig. 8: Distribution of firmware based on the version of Buildroot they are built with.

But should we only blame the automated tools? The answer is clearly no. The vendors largely use old versions of Buildroot even newer versions are available for years, just like what they did to Linux kernels. Table XIII shows the gap between the release time of Buildroot and the use time by vendors. On average, the gap is 5 years. Even the smallest gap is about 2.5 years. This amplified the delay of the availability of attack mitigations. So essentially, the lateness in integrating attack mitigations by the automated tools and the use of older automated tools by embedded vendors together lead to an barrier to the adoption of attack mitigations.

B. Massive Reuse of Binaries

We find that embedded vendors largely reuse binaries across products. First, the same vendor often runs the same group of binaries on different devices. Table XI shows the number of unique binaries from each vendor. On average, only 8.9% of the binaries are unique. In other words, the same binary is reused for 11 firmware. Second, the binaries can also frequently propagate across vendors. Fig. 16 in the Appendix presents the number of unique binaries shared by multiple vendors. Over 8,000 binaries are used by 2 vendors and some binaries are even used by 10 vendors. The heatmap in Fig. 17 in the Appendix gives more details about how frequently vendors borrow binaries from each other.

Reusing binaries across products or even vendors is understandable since these binaries have proven reliability. But how exactly the reuse of binaries affects attack mitigations? To answer this question, we measure the adoption of attack mitigations in binaries reused by multiple vendors. In total, there are 11,232 such binaries. The binaries present a significantly lower adoption rate in most of the mitigations, compared to the results with all binaries considered (compare Table XVII and Table V). That means the propagation of those binaries brings harm to the overall adoption of attack mitigations. Even more worrisome is that the harm will continue until those binaries get rebuilt and redistributed.

C. Cost of the Mitigations

The mitigations can bring extra cost, becoming a possible reason affecting their adoptions. To quantify the cost, we

[5]We intentionally avoid interaction with vendors to prevent ethical issues.

TABLE XI: Number of unique binaries from each vendor (k). The uniqueness of a binary is defined by its MD5.

Vendor	TENVIS	Haxorware	Cerowrt	AT&T	Camius	GOCloud	Actiontec	Buffalo	360	Phicomm	Mercury	Zyxel	CenturyLink	Digi	Supermirco	AVM	MikroTik	OpenWrt	Belkin	NETCore	RouterTech	Dlink	Tenda	TomatoShibby	Linksys	TRENDnet	Moxa	QNAP	Tp-Link-zh	Ubiquiti	Tp-Link-en	NETGEAR	ASUS	Synology	**Average**
Total	0.9	0.2	0.2	0.4	0.6	0.5	0.9	0.4	0.5	0.5	1.9	1.5	0.8	1.5	1.5	5.0	4.3	191.2	7.9	10.2	25.8	15.9	33.6	127.8	17.1	15.3	32.0	296.0	65.7	204.7	76.3	173.9	273.2	1375.4	87.2
Unique	0.1	0.2	0.2	0.2	0.2	0.3	0.3	0.3	0.4	0.5	0.5	0.5	0.7	0.9	0.9	1.8	2.1	2.9	3.4	3.7	4.0	4.7	5.8	5.8	7.0	7.8	12.8	12.8	15.8	20.5	23.6	29.2	29.7	64.9	7.8
Ratio (%)	11.1	100	100	50.0	33.3	60.0	33.3	75.0	80.0	100	26.3	33.3	87.5	60.0	60.0	36.0	48.8	1.5	43.1	36.3	15.5	29.6	17.3	4.5	40.9	50.9	40.0	4.3	24.1	10.1	30.9	16.8	10.9	4.7	8.9

TABLE XII: Cost of attack mitigations on SPEC CPU2006. From left to right, the columns show the **accumulative overhead** after we enable the mitigations, one after another.

Overhead	**NX**	**Canary**	**PIE**	**RELRO**	**Fortify**
Storage	0	6.7%	11.5%	17.3%	17.3%
Memory	0	0	0	0	0
Runtime	0	6.6%	8.45%	10.7%	10.9%

perform an evaluation of the storage/memory/performance overhead of user-space mitigations on SPEC CPU2006, using a Raspberry PI-4B board as the device (Broadcom BCM2711 Quad-core Cortex-A72 with 8GB LPDDR4-3200 SDRAM).

Table XII shows the evaluation results. Overall, applying the mitigations together has no observable overhead on memory usage but introduces a 10.9% and 17.3% overhead on performance and binary size. Specifically, mitigations including Stack Canaries, PIE, and RELRO incur observable overhead in both performance and binary size. Fortify Source, in contrast, brings a lightweight performance overhead without increasing the binary size. We believe these types of overhead may impede the vendors from adopting the mitigations. However, whether that indeed happens needs confirmation with the vendors, which we intentionally avoid for ethical concerns.

Summary: Utilization of old building tools and massive reuse of existing binaries are contributors to the lack of attack mitigations in embedded binaries. Cost of the attack mitigations may also potentially impede their adoption.

VIII. Threats to Validity

A. Representativeness of Dataset

Similar to other sampling-based studies, our study can present findings biased towards the collected dataset. We considered this threat and extended two efforts to reduce the threat. First, we attempted to cover all vendors that are popular or included in previous studies. Second, we enumerate the firmware images that are publicly available from each vendor. In the end, we collected 18k firmware images from 38 vendors. The number of firmware images and the list of vendors are comparable to existing large-scale studies on embedded security [22], [20], [17], [14], [13].

TABLE XIII: Gap between the release time and the use time of Buildroot (months). **Gap Range** shows the range of the gaps for all firmware from the same vendor.

Vendor	**Number**	**Gap Average**	**Gap Range**
360	1	50.0	[50, 50]
Belkin	2	31.5	[29, 32]
Linksys	20	73.0	[40, 104]
Netcore	8	72.4	[44, 98]
TRENDnet	9	48.9	[8, 98]
Tenda	57	47.6	[17, 104]
TomatoShibby	23	32.0	[26, 54]
Tp-Link-zh	71	52.6	[29, 98]
Tp-Link-en	158	63.7	[28, 89]
Dlink	6	58.9	[40, 80]
NETGEAR	259	67.9	[22, 104]
ASUS	59	51.0	[31, 66]
Hikvision	15	33.1	[24, 38]
Ubiquiti	2	64.0	[62, 66]
Average	49.3	60.0	-

B. Imbalance in Dataset

The dataset we collected is not perfectly balanced, which may harm our findings. First, not every vendor has the same amount of data involved. Vendors like OpenWrt and NETGEAR contributed a large portion of the data, while other vendors like Cerowrt and Haxorware only provided a tiny part. Therefore, our generic findings may overfit the dominant vendors. To this end, we break down the results to each vendor in most analyses, which helps raise awareness of the overfitting results. Overall, our generic findings broadly align with the breakdown results. Second, given a mitigation, the applicable data samples are not evenly distributed over time. This can create abnormal points in the analysis of evolution trends. For instance, QNAP released a large number of binaries with Stack Canaries from 2016 to 2018, causing a sudden leap in the evolution trend (recall §V-B). To mitigate the threat, we revisit all the points that appear to be outliers, followed by clarifying the impact of data imbalance (see §V-B and §VI-B).

C. Reliability of Mitigation Identification

First, obfuscations can affect our identification of attack mitigations. For instance, encoding strings can mislead our

detection of Stack Canaries and Fortify Source. In contrast, destroying symbols can disrupt the detection relying on indicator functions. However, we envision that obfuscations should not have affected our study. First, we manually checked many binaries and did not observe obfuscations. Second, the firmware we collected is exclusively from mainstream, benign vendors who have fewer motivations to obfuscate the code.

Second, we rely on static approaches to identify mitigations, which can raise two problems. First, some binaries may not be used at all, so their results do not matter. We did not exclude such binaries. In this regard, our study covers a superset of truly security-relevant binaries. Second, some mitigations can be affected by runtime configurations. For instance, KASLR can be disabled by setting the `nokaslr` parameter when booting the kernel [27]. We, without knowing what happens at runtime, cannot exclude such mitigations. We believe the two problems should not affect our findings much. Based on our study, the adoption rates of attack mitigations are extremely low, which shall still hold even considering a subset of the data and excluding the falsely identified mitigations.

Third, the tools we reuse to help identify may have reliability issues. For instance, FIRMADYNE can extract incomplete kernel data, and Vmlinux-to-ELF can miss recovering kernel functions. Both will hurt our identification of kernel-level mitigations. We realize this threat, but we consider the reliability of existing tools out of this paper's scope.

IX. Related Work

A. Study of Threats to Embedded Devices

Past research has launched many attempts trying to understand the threats faced by embedded devices. Alrawi et al. [8] propose a modeling methodology to systematize the security of home-based IoT devices from the dimensions of attack vectors, mitigations, and stakeholders. They further evaluated 45 open-source or on-market home-based IoT devices, which confirms security issues discussed by their study. Inspired by the study and the evaluation, the authors eventually propose a list of mitigations to address the related security issues. Falling into the category of studying security mitigations, they focus more on discussing the possible mitigations against the attack vectors instead of the adoption of the mitigations in the wild.

Costin et al. [17] present a large-scale study to measure security vulnerabilities in 32k firmware images running on embedded devices. They designed and implemented a distributed architecture to statically measure similarities between firmware images, using a correlation engine. The study discovered 38 unknown vulnerabilities from 693 firmware images. The study also unveils that vulnerabilities from known affected devices can "propagate" to other devices. Similarly, Feng et al. [22] propose and implement Genius, a bug search engine based on features in the control-flow graph. Evaluating Genius on a dataset of 33,045 firmware, the authors discovered 38 potentially vulnerable firmware images from 5 vendors. The two studies and ours all rely on static analysis on a large corpus of firmware to understand embedded security. However, they focus on vulnerabilities while we focus on attack mitigations.

Chen et al. [13] develop FIRMADYNE, an automated, dynamic firmware analysis system, to run firmware binaries through full system emulation and an instrumented kernel. Leveraging FIRMADYNE, they run 74 exploits on 9,486 firmware images. The results unveil that firmware images are largely vulnerable to existing vulnerabilities and exploits: 887 of the firmware images supporting at least 89 distinct products can be affected by one or more of the exploits. Our study complements this work by understanding the adoption of mitigations against those exploits.

B. Study of Attack Mitigations for Embedded Devices

There also exist studies on attack mitigations in embedded devices. The most closely related one to our study is presented in [35]. The authors evaluate the availability of ASLR, Non-executable Stack, RELRO, and Stack Canaries on 28 popular home routers with either ARM or MIPS architecture. The study presents some similar observations to ours. For instance, it finds that the adoption rate of Stack Canaries is extremely low, and NX Stack is more likely to be applied to ARM binaries than MIPS binaries. Compared to this study, ours has a much larger scope and a much higher depth, bringing more systematic insights towards improving the situation.

Other studies in this line focus more on the challenges in applying attack mitigations to embedded devices. Thompson and Zatko [34] explore why MIPS devices usually miss applying NX Stack, as we explained in §V. Abbasi et al. [7] investigate the challenges faced by embedded devices to adopt attack mitigations. They found that many embedded devices, particularly the low-end ones, often lack the hardware and OS support needed by the mitigations. In our study, we concentrate on Linux-based devices where the hardware and the OS are less restricted. We aim to understand that, when objective restrictions like those discussed in [34] do not exist, how often the mitigations will be adopted.

C. Tools for Mitigation Measurement

At the time we conduct the study, many existing tools, including Checksec [12], Hardening-Check [36], and Pwntools [31], can help detect user-space mitigations. Checksec [12] is a bash script designed to test standard security properties of ELF files. It additionally provides the feature of detecting kernel-level mitigations in running systems. Hardening-Check [36] is another tool providing similar features as Checksec. Pwntools [31] is a CTF framework and exploit development library. It provides functionality to check the status of the above security features applied in ELF binaries. We follow these tools to develop many of our detection strategies, but also extend them. First, we extend the mitigation detection to handle cases like Stack Canaries in statically linked, stripped binaries. Second, we add new supports to detect mitigations applied in the kernel without running it in the actual device.

X. Conclusion

This paper measures the adoption of standard attack mitigations in embedded devices. It shows that attack mitigations are largely missing even on devices where the needed hardware/OS supports are fully available. The findings also complement previous research that ties the absence of mitigations to the lack of hardware/OS supports. By inspecting the evolution over time, the study unveils that the situation does not improve

in the past decade, casting a worrisome prediction about the upcoming future where embedded devices will explode. On the positive side, the study identifies a set of doings hurting the adoption of attack mitigations, which bring insights towards improving the current practice.

Acknowledgments

We thank our shepherd Avesta Sasan and the anonymous reviewers for their feedback. This project was supported by National Science Foundation (Grant#: CNS-2031377), Office of Naval Research (Grant#: N00014-17-1-2787; N00014-17-1-2788), and DARPA (Grant#: D21AP10116-00). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] "'amnesia:33' vulnerabilities in tcp/ip stacks expose millions of devices to attacks," https://www.securityweek.com/amnesia33-vulnerabilities-tcpip-stacks-expose-millions-devices-attacks.

[2] "Firmware scraper," https://github.com/firmadyne/scraper.

[3] "Relro: make the relocation section that are used to resolve dynamically loaded functions read-only," https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-54839.html.

[4] "Non-executable stack (nx): marks memory regions as non-executable," https://insecure.org/sploits/non-executable.stack.problems.html, 1998.

[5] "Fortify source: provides buffer overflow protection to some memory and string functions," https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html, 2004.

[6] "Linux kernel defence map," https://github.com/a13xp0p0v/linux-kernel-defence-map, 2018.

[7] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 31–46.

[8] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 1362–1380.

[9] S. Andersen and V. Abella, "Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention," 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[10] Bulba and Kil3r, "Bypassing stackguard and stackshield," Phrack Magazine, May 2000, http://phrack.org/issues/56/5.html.

[11] P. Burton, "Mips: Use per-mm page to execute branch delay slot instructions," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=432c6bacbd0c16ec210c43da411ccc3855c4c010, 2016.

[12] checksec, "checksec : a bash script to check the properties of executables (like pie, relro, pax, canaries, aslr, fortify source)," https://github.com/slimm609/checksec.sh, 2020.

[13] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, vol. 1, 2016, pp. 1–1.

[14] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[15] 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 *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 289–303.

[16] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 309–326.

[17] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 95–110.

[18] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[19] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of iot malware," in *Annual Computer Security Applications Conference*, 2020, pp. 1–16.

[20] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, 2018.

[21] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 463–478.

[22] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[23] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and securing device vulnerabilities through automated bug report analysis," in *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[24] Gilad David Maayan, "The iot rundown for 2020: Stats, risks, and solutions," https://securitytoday.com/articles/2020/01/13/the-iot-rundown-for-2020.aspx, 1 2020.

[25] B. Group, "Buildroot: A simple, efficient and easy-to-use tool to generate embedded linux systems through cross-compilation," https://buildroot.org/, 2021.

[26] M. Guri, "Aslr - what it is, and what it isn't," https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/#:~:text=Address%20Space%20Layout%20Randomization%20(ASLR,in%20a%20process's%20address%20space., 2015.

[27] R. Hat, "Kernel address space randomization red hat enterprise linux," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_security_guide/sect-virtualization_security_guide-guest_security-kaslr.

[28] C. Heffner, "Binwalk - firmware analysis tool," https://github.com/ReFirmLabs/binwalk, 2021.

[29] IoT Inspector Research Lab, "Advisory: Multiple issues in realtek sdk affects hundreds of thousands of devices down the supply chain," https://www.iot-inspector.com/blog/advisory-multiple-issues-realtek-sdk-iot-supply-chain, 2021.

[30] Marin, "vmlinux-to-elf," https://github.com/marin-m/vmlinux-to-elf.

[31] pwntools, "pwntools : a ctf framework and exploit development library," https://docs.pwntools.com/en/stable/index.html, 2016.

[32] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 114–138.

[33] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[34] P. Thompson and M. Zatk, "Linux mips - a soft target: past, present, and future," https://cyber-itl.org/assets/papers/2018/Linux_MIPS_missing_foundations.pdf, 2018.

[35] P. Thompson and S. Zatko, "Build safety of software in 28 popular home routers," https://cyber-itl.org/assets/papers/2018/build_safety_of_software_in_28_popular_home_routers.pdf, 2018.

[36] Ubuntu, "Hardening checks - check binaries for security hardening features," http://manpages.ubuntu.com/manpages/trusty/man1/hardening-check.1.html, 2021.

Appendix

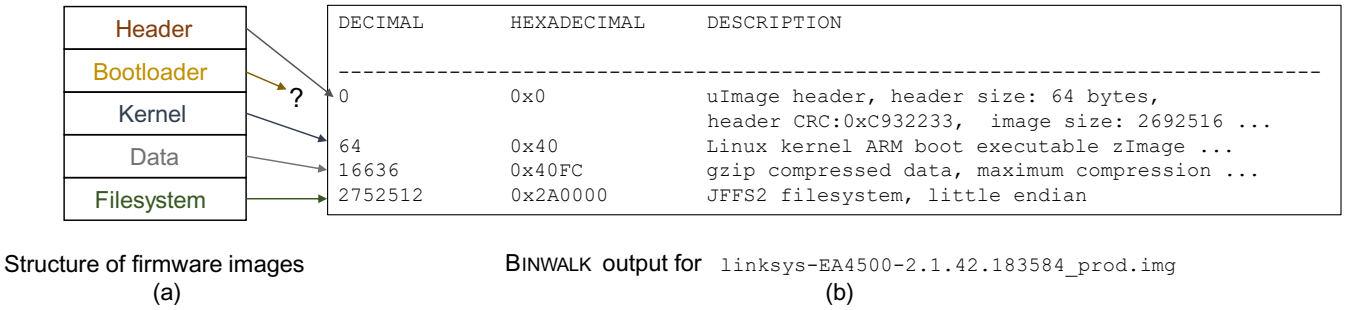


Fig. 9: An example of Linux-based firmware image.

```
;function start
30: ldr  r3             , data_101a0
34: ldr  r4             , data_101a4
38: push {r0, r1, r2, lr}
3c: ldr  r6             , data_101a8
40: add  r4             , pc     , r4  ; {data_6a28f, *** stack smashing detected ***:...}
44: ldr  r5             , [pc, r3]
48: mov  r0             , r4           ; {data_6a28f, *** stack smashing detected ***:...}
4c: bl   sub_10110
......
9c: bl   sub_64d94
;function end
```

Listing 1: The `stack_chk_fail` function in a statically-linked binary.

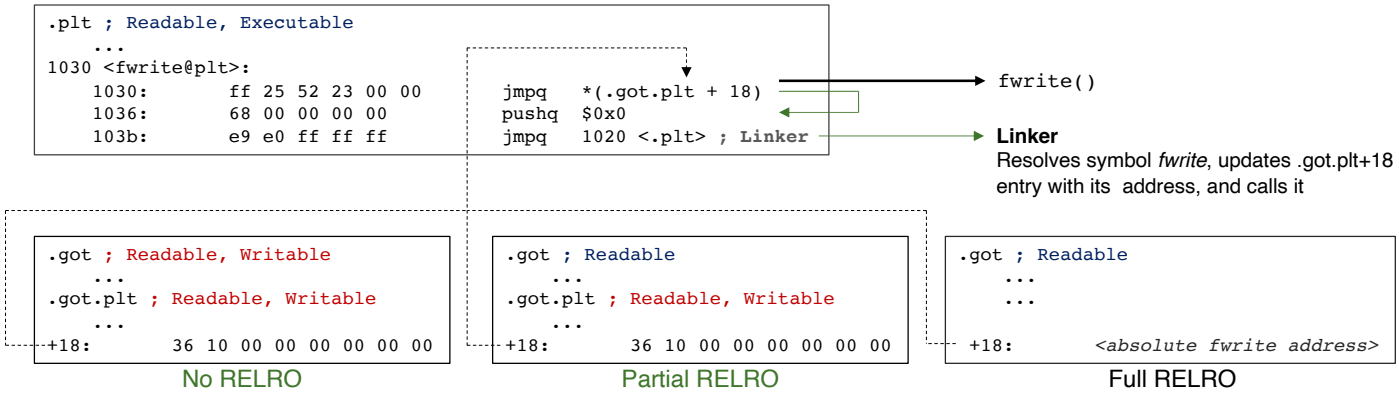


Fig. 10: Binary with no/partial/full RELRO.

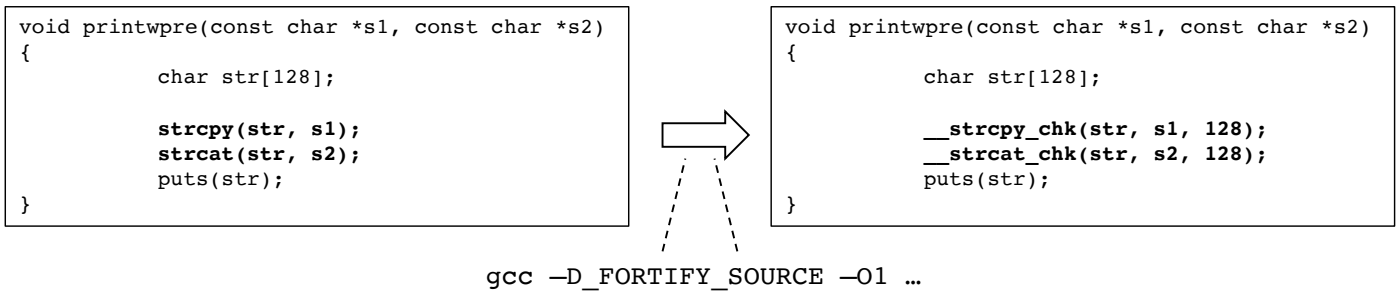
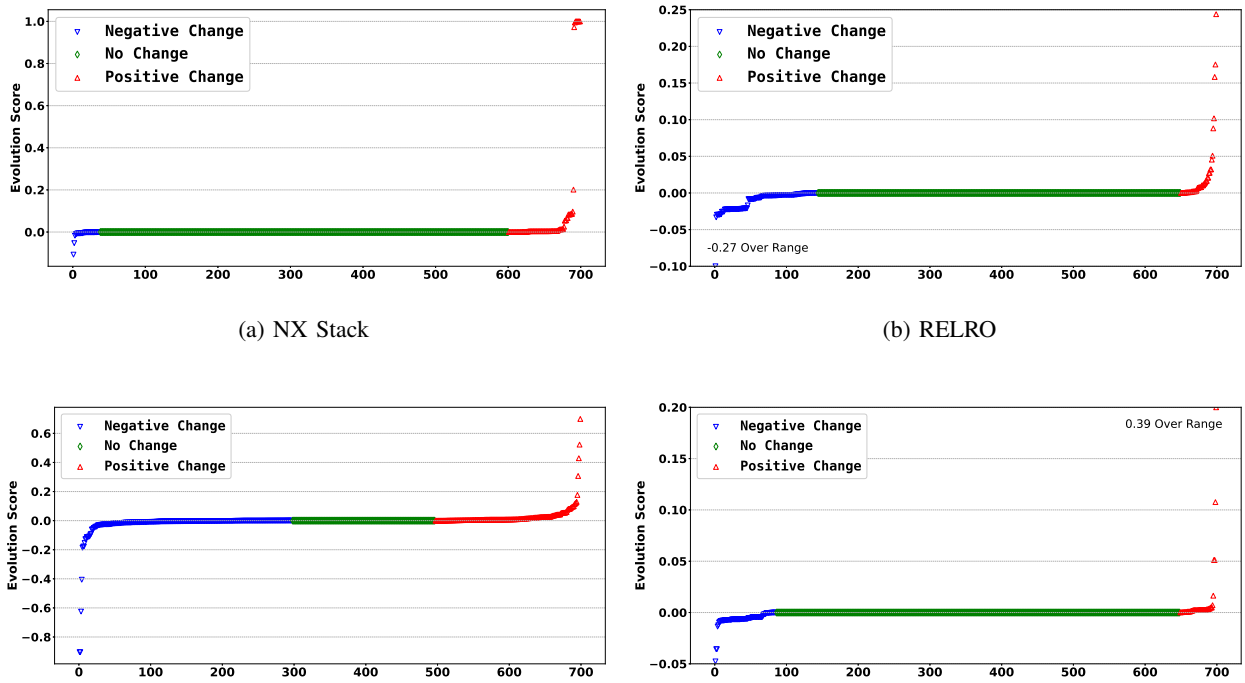


Fig. 11: Replacement of dangerous libc functions with safer versions by Fortify Source.



(a) NX Stack

(b) RELRO

(c) PIE

(d) Fortify

Fig. 12: Evolution score of individual firmware in the adoption of different mitigations. Each point represents a firmware with multiple versions. The firmware is sorted based on the evolution score.

TABLE XIV: Breakdown results of evolution score of firmware with multiple versions. `[scr1, scr2]` in each cell means the evolution score for all firmware of the same vendor ranges from `scr1` to `scr2`.

Vendor	# of Firmware	Canary	RELRO	NX	Fortify	PIE
360	1	[7.4, 7.4]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
AT&T	1	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
Buffalo	1	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
Phicomm	1	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
TENVIS	1	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
TRENDnet	2	[0, 0.3]	[0, 2.8]	[0, 0]	[0, 0]	[0, 2.8]
Zyxel	2	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
AVM	3	[0, 0.3]	[0, 5.1]	[-0.06, 0]	[0, 0]	[0, 5.1]
Moxa	3	[0, 10.1]	[0, 15.8]	[0, 20.1]	[0, 10.8]	[0, 15.8]
Mercury	5	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
TomatoShibby	5	[0, 0.02]	[0, 0.2]	[0, 5.6]	[0, 0]	[0, 0.2]
Netcore	6	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
RouterTech	7	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
MikroTik	8	[-0.01, 0]	[-0.2, 0]	[0, 0.01]	[-0.1, 0]	[-0.2, 0]
Belkin	12	[0, 0]	[0, 3.2]	[-10.7, 100]	[0, 0]	[0, 3.2]
QNAP	12	[-15.1, -13.8]	[-3.0, -2.2]	[-0.2, 0.08]	[-0.7, -0.2]	[-3.0, -2.2]
Linksys	14	[-0.01, 2.4]	[-0.1, 24.4]	[0, 99.5]	[0, 0]	[-0.1, 24.4]
Dlink	19	[-3.6, 0.8]	[0, 0.2]	[-0.4, 100]	[-3.6, 0]	[0, 0.2]
Tenda	61	[-0.1, 7.8]	[-0.8, 4.5]	[0, 0]	[0, 0]	[-0.8, 4.5]
Tp-Link-en	72	[-0.03, 7.2]	[-0.3, 0.1]	[-0.3, 0.09]	[-0.05, 5.1]	[-0.3, 0.1]
Tp-Link-zh	76	[-7.2, 7.6]	[-1.7, 2.1]	[-0.7, 100]	[0, 0.7]	[-1.7, 2.1]
Synology	94	[-3.8, 40.1]	[-2.3, 17.5]	[0, 0.5]	[-1.4, 38.6]	[-2.3, 17.5]
ASUS	143	[-2.0, 1.0]	[-2.1, 1.0]	[-0.9, 9.7]	[-0.8, 0.5]	[-2.1, 1.0]
Netgear	150	[-4.8, 60.3]	[-27.4, 10.2]	[-5.3, 8.3]	[-4.8, 1.6]	[-27.4, 10.2]

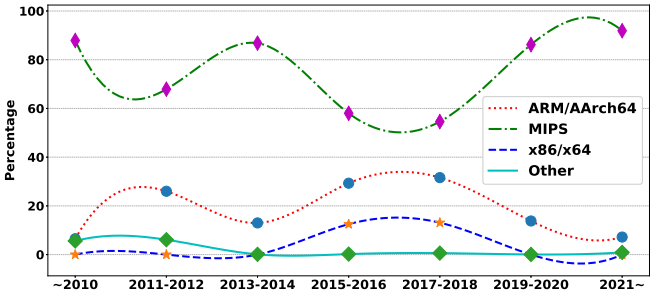


Fig. 13: Distribution of binaries based on their architectures. All binaries released before 2010 are aggregated into ∼2010.

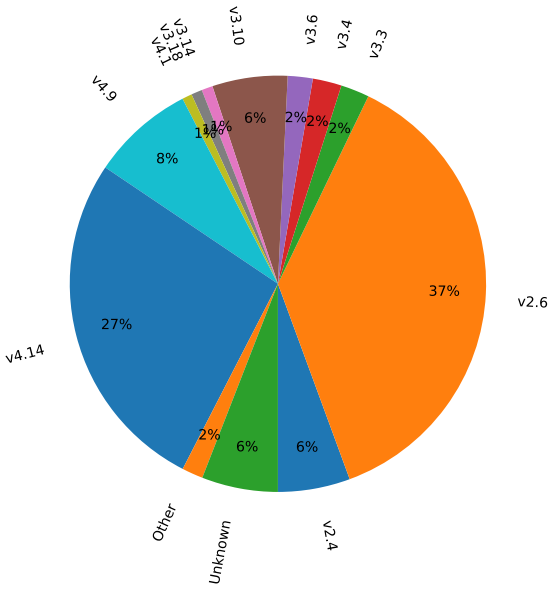


Fig. 14: Distribution of different kernel versions.

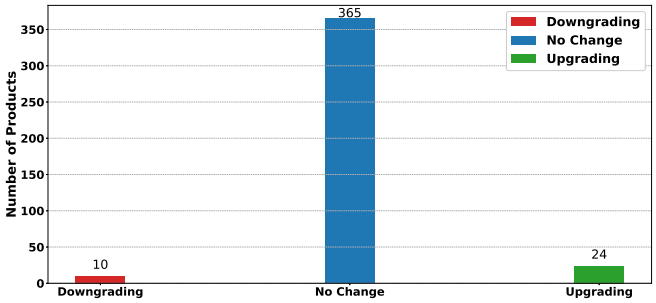


Fig. 15: Changes of kernels running on different versions of the same firmware. `Downgrading`, `No Change`, `Upgrading` respectively indicate the number of firmware with kernels downgraded to older versions, with kernels remained at the same version, and with kernels upgraded to newer versions.

TABLE XV: Types of device covered in our study

Router, Web Camera, Network Port, Network Switch, Network Storage, Network Access Point, Repeater, Adapter, WIFI Extender, WIFI System, WIFI Bridge, Controller, Video Recorder, Radio, Mother Board, Gateway, Media Connector, Printer, Firewall Modem

TABLE XVI: Availability of attack mitigations in different versions of Buildroot.

Version	Default Kernel	Canary	SC[1] Dependency	RELRO	Fortify	PIE
2021-02	v5.10	✓	✓	✓	✓	✓
2020-11	v5.4	✓	✓	✓	✓	✓
2019-11	v4.19	✓	✓	✓	✓	✓
2018-11	v4.16	✓	✓	✓	✓	☒
2017-11	v4.13	✓	✓	☒	☒	☒
2016-11	v4.8	✓	✓	☒	☒	☒
2015-11	v4.3	✓	✓	☒	☒	☒
2014-11	v3.17	✓	✓	☒	☒	☒
2013-11	v3.11	✓	✓	☒	☒	☒
2012-11	v3.6	✓	☒	☒	☒	☒
2011-11	v3.1	✓	☒	☒	☒	☒
2010-11	v2.6	✓	☒	☒	☒	☒
2009-11	v2.6	✓	☒	☒	☒	☒

[1] "SC" is short for Stack Canaries.

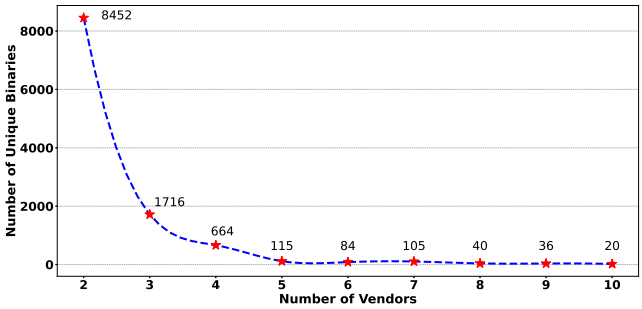


Fig. 16: Number of unique binaries reused by multiple vendors. For example, the point at `x = 2` means 8,452 unique binaries are reused by 2 vendors (without considering binaries reused by more than 2 vendors).

TABLE XVII: Adoption rates of user-space mitigations by binaries shared by multiple vendors (%).

# of Binaries	Canary	RELRO	NX	Fortify	PIE
11232	15.4	9.2	46.0	15.3	34.5

TABLE XVIII: Evolution of individual binaries in adopting attack mitigations (breakdown results). In each cell, `+x / -y` indicates `x` binaries have the mitigation added and `y` binaries have the mitigation dropped; "-" means no change.

Vendor	ELF	Canary	RELRO	NX	Fortify	PIE
360	118	-	-	-	-	-
AT&T	190	-	-	-	-	-
Buffalo	9	-	-	-	-	-
Phicomm	147	-	-	-	-	-
TENVIS	36	-	-	-	-	-
TRENDnet	409	+1 / -0	+6 / -0	-	-	-
Zyxel	154	-	-	-	-	-
AVM	358	+6 / -0	+20 / -0	-	-	+11 / -0
Moxa	107	-	-	-	-	-
Mercury	232	-	-	-	-	-
TomatoShibby	735	-	-	-	-	-
Netcore	296	-	-	-	-	-
RouterTech	1086	-	-	-	-	+65 / -0
MikroTik	1100	-	-	-	-	-
Belkin	688	-	-	+69 / -0	-	-
QNAP	9192	+11 / -0	+0 / -39	-	+26 / -0	+26 / -0
Linksys	3420	+1 / -0	+3 / -0	+171 / -0	-	+4 / -3
Dlink	2399	-	-	+236 / -0	-	-
Tenda	7266	+8 / -0	-	-	-	+17 / -0
Tp-Link-en	9728	-	-	-	-	+0 / -3
Tp-Link-zh	10110	+2 / -2	-	+221 / -0	-	+0 / -14
Synology	179949	+13 / -233	+177 / -9	+139 / -0	+19 / -88	+353 / -0
ASUS	31581	+9 / -0	+0 / -1	+6 / -1	-	+1 / -0
Netgear	20289	+387 / -49	+117 / -73	+164 / -218	+16 / -16	+333 / -95

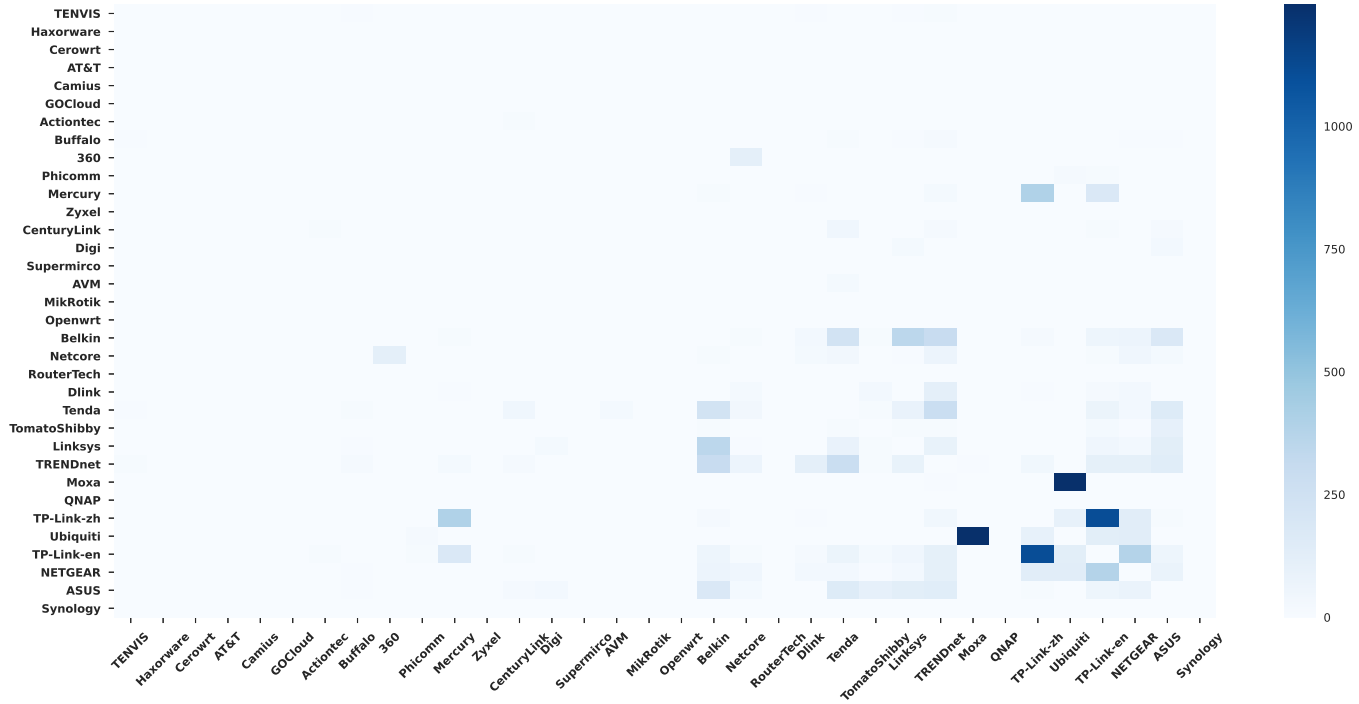


Fig. 17: Heatmap showing how many binaries vendors borrow from each other.