Since first identified by Anderson [14] in the 70s, memory corruption vulnerabilities are still among the top three of the CWE SANS most dangerous software errors [59]. Even now, nearly 400 memory corruption errors have been discovered in CVEs [115] in the past 10 years. Memory corruption errors are especially common in applications written in type-unsafe languages like C and C++, due to the pointer operations supported in these languages. The ability to manipulate pointers provides an efficient way to design applications and systems, but using it incorrectly often leads to unintentional modification of related memory content, enabling an attacker to exploit such a vulnerability by either compromising sensitive data or hijacking the control flow. It is hard to convince developers to switch to type-safe languages, not only because of performance, but also because of backward compatibility reasons. The problem of improving the security of applications or systems written in C/C++ has been recognized by the community for a long time, but previous solutions often introduce non-acceptable overheads, and many require source information, which is often unavailable for legacy binaries.

On the attacker side, in order to compromise sensitive objects, it is essential to know the exact location of the sensitive objects at runtime. Attackers often acquire such information via static/dynamic analysis of the vulnerable binary, but this was before Address Space Randomization (ASLR) was deployed in systems. With ASLR enabled, address spaces are randomized at load time to destroy the attacker’s prior knowledge of the vulnerable binary, causing crashes due to failed "guesses" of the sensitive object location by the attacker.

As part of the arms-race, today’s attackers try to exploit memory disclosure vulnerabilities and use information leakage attacks to bypass ASLR. Since ASLR only randomizes the address space at loadtime, once the attacker retrieves the location of
the sensitive objects, the attack procedure is no different from prior to ASLR. For example, JIT-ROP attacks [160] begin with a memory disclosure attacks: the attacker leaks the content of the code space and then launches code-reuse attacks. Memory disclosure attacks most commonly rely on arbitrary memory reads or writes to leak the location of sensitive objects, but they are not the only possibilities. With side-channel attacks [153], such as fault or timing side-channels, the attacker can learn properties of the binary without reading/writing it. For example, blind-ROP (BROP) [26] attacks rely on the crash-recovery mechanisms used by modern servers to retrieve the content of the code space with side-channel attacks.

In this dissertation, we argue that address space runtime randomization can be effective against memory corruption attacks with general applicability to different memory spaces: heap space, stack space and code space. Our investigation focuses on improving the security guarantees provided by ASLR and prior work in the area, while also improving performance, capabilities, and re-randomization frequency. To substantiate our claims, we present several contributions and demonstrate the viability of the proposed techniques in practice. In particular, we present the first self-contained solution to randomize legacy binaries at runtime without requiring for any hardware, kernel, or source code support.

Concerning the heap space, when the application has its own memory management functions, heap-based defenses either rely on a human supervisor to indicate custom memory allocators and deallocators or only protect memory objects allocated by the system allocator. Our MemBrush is the first tool to detect memory allocation and deallocation functions automatically in legacy binaries with high accuracy by using dynamic taint analysis. Our evaluation on real-world applications shows that MemBrush can archive high accuracy and further detect the type of the memory allocators and the semantics of their arguments.

Concerning the stack space, previous defenses mainly focused on spatial vulnerabilities and left temporal vulnerabilities exploitable. We propose StackArmor as a comprehensive protection technique to address both spatial and temporal stack vulnerabilities in legacy binaries. StackArmor, relies on binary analysis and rewriting and does not need access to the source code. The evaluation of our x86_64 Linux prototype shows that StackArmor offers better security than prior binary- and source-level approaches, at the cost of modest performance and memory overheads.

To prevent modern code-reuse attacks, we propose CodeArmor, a binary-level system to harden code diversification against all existing read-based (JIT-ROP) and execution-based (BROP) disclosure attacks. Similar to StackArmor, CodeArmor relies on binary analysis and rewriting to handle legacy binaries. CodeArmor virtualizes the code space to decouple code pointer values from the concrete location in code space and periodically remap the concrete code space into different locations at runtime. By using RCU, CodeArmor can provide the lowest remapping latency compared to previous approaches. CodeArmor's honey gadgets can further detect blind code-reuse attacks by design with no false positives.