Q1:None.

Q2:The attackers are “system attackers” who execute and exploit memory bugs in vulnerable applications written in low-level languages like C/C++ on victims’ machines in order to alter the program’s behavior to gain more control, gain privileges, or leak information. The victims are application programmers who write the programs and legitimate users who have the vulnerable applications on their machines. The attacker starts by crafting an invalid pointer which either goes out-of-bounds or becomes dangling, then triggers a memory error by reading or writing through the invalid pointer that facilitates further exploitations. The attacker can make a pointer out-of-bounds when he has control over the index into an array where the bounds check is missing or incomplete, or make a pointer dangling by exploiting an incorrect exception handler which deallocates an object but does not reinitialize the pointers to it.

Q3:(1) Code corruption attack, where the program code in memory is overwritten to the attacker-specified code. (2) Control-flow hijack attack, where a code pointer is corrupted with the address of the malicious payload and loaded into the instruction pointer by an indirect control-flow transfer instruction. The payload can either be shell code injected by the attacker, or existing code that is either an existing function or small instruction sequences chained together to carry out malicious operations. (3) Data-only attack, where neither code nor code pointers are corrupted, but some security critical data in memory, such as configuration data, user identity, or keys, are modified. (4) Information leak, where memory contents which would otherwise be excluded from the output are leaked and used to circumvent probabilistic defenses based on randomization and secrets.

Q4:(1) Memory safety, including spatial safety and temporal safety. Spatial safety protects against out-of-bounds pointers, and can be enforced using pointer-bound methods like “CCured”[1], Cyclone[2] and “SoftBound”[3], or object-bounds methods like “J&K”[4], “CRED”[5], “automatic pool allocation”[6], and “BBC”[7]. Temporal safety protects against use-after-free and double-free vulnerabilities, and can be enforced using special allocators like “Cling”[8], object-based approaches like “Valgrind”[9] and “AddressSanitizer”[10], or pointer-based approaches like “CETS”[11]. Enforcing memory safety stops all memory corruption exploits.

(2) Data integrity enforces an approximation of spatial memory integrity but does not enforce temporal safety or protects against invalid memory reads. It can be enforced by the integrity of “safe” objects like in Yong’s system[12], or the integrity of points-to sets like “WIT”[13] and “BinArmor”[14]. Code integrity and code pointer integrity are both subsets of data integrity. The former enforces that program code cannot be written, which can be achieved if all memory pages containing code are read-only. The later aims to prevent the corruption of code pointers.

(3) Data space randomization (DSR)[15] randomizes the representation of data in memory by encrypting all variables using different keys to probabilistically mitigate all attacks. Address space randomization (ASR), a subset of data space randomization, mitigates control-flow hijacking attacks by randomizing the location of code and data and thus the potential payload address. It includes methods like ASLR[16], PIE[17] and STIR[18]. Between DSR and ASR there is a pointer encryption technique called “PointGuard”[19], which encrypts all pointers in memory and only decrypts them before they are loaded into a register.

(4) Data-flow integrity (DFI)[20] detects the corruption of any data before it gets used by restricting reads based on the last instruction that wrote the read location. Control-flow integrity (CFI)[21], a subset of data-flow integrity, enforces policies regarding indirect control transfers, including methods enforcing dynamic return integrity like stack cookies[22], shadow stacks[23], or RAD[24], and methods enforcing static control-flow graph integrity like in Abadi's work[25].

(5) Non-executable data policy uses the executable bit to make data memory pages non-executable. A combination of Non-executable data and Code Integrity results in the W$\oplus$X policy, stating that a page can be either writable or executable, but not both.

(6) Instruction set randomization (ISR)[26] mitigates the execution of injected code or the corruption of existing code by encrypting it.

Q5:The cost-effectiveness is determined by protection, overhead, and compatibility. A good defense mechanism should provide comprehensive protection and broad compatibility but incur low overhead. (1) Protection. The strength of the protection is determined by the effectiveness of the security policy it enforces, which, in turn, is determined by the attacks it can protect against. The false negative rate (probability of a successful attack) should be as low as possible and false alarms (e.g., unnecessary crashes) are unacceptable. (2) Overhead, including performance overhead and memory overhead. To measure performance overhead, both CPU-bound and I/O-bound benchmarks can be used, and the former is recommended. Memory overhead can be introduced by inline monitors or shadow memory but is less of a concern. (3) Compatibility, including source compatibility, binary compatibility, and modularity support. An approach is source compatible if it does not require application source code to be manually modified to profit from the protection. Binary compatibility allows compatibility with unmodified binary modules. Supports for modularity means that individual modules are handled separately.

Q6:The main reasons are performance and compatibility. Techniques introducing an overhead larger than roughly 10% are not used in practice. Examples are SoftBound[3], CETS[11], BBC[7], WIT[13], DSR[15], DFI[20], PIE[17], Shadow stack[23], and CFI[21]. Compatibility issues also prevent the deployment of many techniques. Techniques that are not source compatible (e.g. PointGuard[19], CCured[1], Cyclone[2]) require human intervention and are considered unscalable and impractical. Techniques that are not binary compatible (e.g., SoftBound[3], CETS[11], WIT[13], DSR[15], DFI[20], CFI[21]) prevents transformed program from linking with unmodified legacy libraries. Techniques that do not have modularity support (e.g., WIT[13], DSR[15], DFI[20], CFI[21]) require consideration of the dependencies between modules, which makes re-usable libraries challenging. Also, some defense mechanisms have become obsolete (e.g., ISR[26]) because the same attack vectors can be defeated by enforcing other more cost-effective policies, some are considered less important because attacks they protect against are not considered as valid threats (e.g., enforcing Control-flow Integrity[21] instead of Data Integrity when considering only hijacking attacks as valid), and some policies simply cannot be enforced without causing false positives, e.g., Code Integrity does not support self-modifying code or JIT compilation.

The paper I choose is “SoK: Eternal War in Memory”[0].

References:

[0] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in SP’13.

[1] G. C. Necula, S. McPeak, and W. Weimer, “CCured: type-safe retrofitting of legacy code,” in POPL’02.

[2] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of C,” in USENIX ATC’02.

[3] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: highly compatible and complete spatial memory safety for C,” SIGPLAN Not.’09.

[4] R. Jones and P. Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs,” Auto. and Algo. Debugging’97.

[5] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in NDSS’04.

[6] C. Lattner and V. Adve, “Automatic pool allocation: improving performance by controlling data structure layout in the heap,” in PLDI’05.

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors,” in USENIX Security’09.

[8] P. Akritidis, “Cling: A memory allocator to mitigate dangling pointers,” in USENIX Security’10.

[9] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in PLDI’07.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in USENIX ATC’12.

[11] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “CETS: compiler enforced temporal safety for C,” in ISMM’10.

[12] S. H. Yong and S. Horwitz, “Protecting C programs from attacks via invalid pointer dereferences,” in ESEC/FSE-11’03.

[13] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with WIT,” in IEEE SP’08.

[14] A. Slowinska, T. Stancescu, and H. Bos, “Body armor for binaries: preventing buffer overflows without recompilation,” in USENIX ATC’12.

[15] S. Bhatkar and R. Sekar, “Data Space Randomization,” in DIMVA’08.

[16] T. PaX, “Address space layout randomization,” 2001. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[17] M. Payer, “Too much PIE is bad for performance,” 2012.

[18] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, “Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code,” in CCS’12.

[19] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard: protecting pointers from buffer overflow vulnerabilities,” in USENIX Security’03.

[20] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in OSDI’06.

[21] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Controlflow integrity,” in CCS’05.

[22] H. Etoh and K. Yoda, “Protecting from stack-smashing attacks,” 2000, http://www.trl.ibm.com/projects/security/ssp.

[23] Vendicator, “Stack Shield: A ”stack smashing” technique protection tool for linux,” 2000.

[24] T. Chiueh and F.-H. Hsu, “RAD: A compile-time solution to buffer overflow attacks,” in ICDCS’01.

[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Controlflow integrity,” in CCS’05.

[26] G. S. Kc, A. D. Keromytis, and V. Prevelakis, “Countering code-injection attacks with instruction-set randomization,” in CCS ’03.