IEICE TRANS. INF. & SYST., VOL.E99–D, NO.8 AUGUST 2016

PAPER Special Section on Security, Privacy and Anonymity of Internet of Things

BFWindow: Speculatively Checking Data Property Consistency against Buffer Overflow Attacks

Jinli RAO[†], *Nonmember*, **Zhangqing HE**[†a)], *Student Member*, **Shu XU**[††b)], **Kui DAI**[†††], *and* **Xuecheng ZOU**[†], *Nonmembers*

SUMMARY Buffer overflow is one of the main approaches to get control of vulnerable programs. This paper presents a protection technique called *BFWindow* for performance and resource sensitive embedded systems. By coloring data structure in memory with single associate property bit to each byte and extending the target memory block to a *BFWindow(2)*, it validates each memory write by speculatively checking consistency of data properties within the extended buffer window. Property bits are generated by compiler statically and checked by hardware at runtime. They are transparent to users. Experimental results show that the proposed mechanism is effective to prevent sequential memory writes from crossing buffer boundaries which is the common scenario of buffer overflow exploitations. The performance overhead for practical protection mode across embedded system benchmarks is under 1%.

key words: *embedded system security, buffer overflow, data structure coloring, data property consistency, speculatively checking*

1. Introduction

With advances of sensors, embedded computing and communication technologies, internet has spread from persons to things known as Internet of Things(IoT) [1], [2] which enables an "always-connected" paradigm for our society. Along with the benefits from IoT, challenges such as infrastructure design, data/service management and security have raised attentions [3]–[5]. Information security attacks are growing with the widely used of IoT devices. It leads security become an important metric for system design as well as function, speed, area and power [6], [7]. In order to achieve high performance and low-level hardware controllability, embedded software normally adopts unsafe languages, notably C and C++. Unfortunately, C/C++ are weak type languages and don't apply data structure boundary checking for memory access. This allows manipulations of memory contents with arbitrary data pointer casting and dereference which natively facilitates memory attacks.

Buffer overflow is one of the top software memory exploitation threats [8], [9]. Though numbers of protection methods have been proposed, it still dominates due to security limitations and usability poorness of existing methods.

Security limitations: (1) Incomplete protection space: proposed methods target defending partial attacks such as corruptions of function pointers and return address, which can be easily bypassed by new attack approaches. (2) Delayed attack detection: most protection methods detect attacks when the corrupted data is referred, but the time difference between malicious data injection and detection may cause system suffering information leakages or hardware damages as discussed in Sect. 3.2.

Usability poorness: (1) Performance overhead: software based solutions degrade performance significantly and designers are reluctant to adopt them. (2) Deploy burden: hardware based solutions need deep changes of system design and also rely on software explicitly programming the protection functionality.

In order to bridge the gap among security, performance and usability, this paper presents an efficient compiler and hardware based method called *BFWindow* for embedded systems. Compiler generates program data structure property map and hardware validates memory writes by checking property consistency of an extended buffer window including the target data. It efficiently prevents sequential memory writes from crossing data structure boundary which is known as buffer overflow. This new angle of view permits us to contribute in following areas:

Integrity: *BFWindow* eliminates buffer corruptions caused by sequential memory writes. Malicious data can't be injected and propagated to other variables through buffer boundary crossing. Moreover, it performs validation before memory targets being really accessed which detects attacks immediately after attack attempts and maintains a secure runtime environment for programs.

Low Overhead: Hardware based checker resides in memory system for background validation. It releases processor resource required by instruction based solutions and minimizes performance penalty. As it's decoupled from processor pipeline, it also simplifies hardware implementation.

Usability: Long term programming habits hinder programmers following new software design rules and changes of legacy code also bring risk to the system. *BFWindow* takes advantages of compiler and hardware which are transparent to programmers and legacy code. Neither constraints to programming style nor patches to legacy code are needed for security upgrading which makes it easy for deployment.

Manuscript received November 30, 2015.
Manuscript revised March 28, 2016.
Manuscript publicized May 31, 2016.

[†]The authors are with School of Optical and Electronic Information, Huazhong University of Science and Technology, Wuhan 430074, China.
[††]The author is with Science and Technology on Information Assurance Laboratory, Beijing, China.
[†††]The author is with Institute of National Network Security and Information, Peking University, Beijing, China.
a) E-mail: ivan_hee@126.com
b) E-mail: 18010005098@189.cn (Corresponding author)

DOI: 10.1587/transinf.2015INP0003

The rest of the paper is organized as follows. Section 2 goes over available buffer overflow protection methods. Section 3 describes models of target system and attacks from hardware and software perspectives. Proposed protection mechanism and implementation are detailed in Sect. 4 and 5 respectively. Section 6 presents security and performance experiments results and Sect. 7, the conclusion.

2. Related Work

Researches about buffer overflow attack approaches and protection methods are widely developed since 1960s [27], [28], [30]. Libsafe [31] targets the common vulnerable library functions by reimplementing them with buffer overflow protection capability, but it cannot protect homebrew code. Compiler is heavily exploited to automatically strengthen code with buffer overflow detections. StackGuard [13] and StackShield [14] take advantages of compiler to insert code into function prologue and epilogue to perform control data protection. StackGuard protects the function return address through a canary guard word while StackShield duplicates the function return address with the assumption that attacker may not override them at the same time. However, both StackGuard and StackShield target function return address only and leave function pointers as well as local variables vulnerable to attacks like "return-to-libc" [15]. ProPolice/SSP [16] enhances StackGuard with local variable protection by reordering buffers to the opposite direction of buffer growing, but adjacent buffers are left to be attackable. CCured [10] and CRED [11] use static analysis and runtime boundary checking to prevent buffer access abuse. WIT [12] uses points-to analysis to get the full map of pointer usages and employs operating system to maintain data coloring table in a separate virtual memory space. Its mechanism is similar to our solution. But the analysis time increases dramatically with source code size and the multiple bits coloring table is visible to attackers which may potentially be overridden.

Hardware methods are proposed for higher efficiency and security. Non-eXecute(NX) bit marks memory sections as non-executable [17] to prevent execution of injected code. Secure Return Address Stack(SARS) is a hardware implementation of StackShield targeting function return address protection [18]. Memory tag technologies [19], [20] implement fine grain control of memory access with associate bits to memory items, but it relies on software to manually implement the protection mechanism which hinders its deployment. [21]–[23] track data flow and strictly avoid suspicious inputs to be used as control data. The implementations need redesign of hardware and may cause software compatibility issues such as function pointer table usages. Recently hardware based boundary checking solutions appear. [24] implements implicit boundary checking while [25], [26] implement explicit boundary checking. A pointer table of (*address*, *base*, *size*) is generated by compiler and maintained by hardware μops or extended instructions. They can effectively eliminate memory errors. But even for the most efficient liner pointer table implementation, it costs 2 times memory space overhead which is not practical for embedded systems.

3. Security Model

3.1 System Security Model

An m bits register-based computing system is illustrated as Fig. 1. Its security domain includes trusted computing contexts and untrusted inputs. Computing contexts include runtime code, data and devices which are either physically untouchable like remote access only or secured by other protection mechanisms like data encryption or anti-tampering chips. And inputs are exposed to attackers which are easily controlled through software provided user interfaces.

The inputs and computing contexts are connected by system state transformation as Fig. 2 with inputs I, outputs O, system architecture visible states M, system function μ, state to function converter f and system state change δ for instruction i. The formal representations are:

$$(O_i, M_{i+1}) = \mu_i(I_i, M_i)\ \{I_i, O_i \in [-2^m, 2^m - 1]\} \quad (1)$$

$$\mu_i = f(M_i) \quad (2)$$

$$\delta_i = M_{i+1} - M_i\ \{\delta_i \in [-2^m, 2^m - 1]\} \quad (3)$$

If the system has incomplete validation of inputs, malicious data can be injected without triggering system alarms.

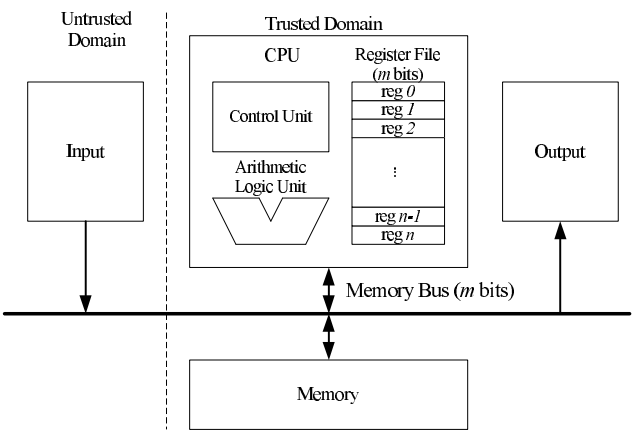


Fig. 1 m bits datawidth computing system diagram

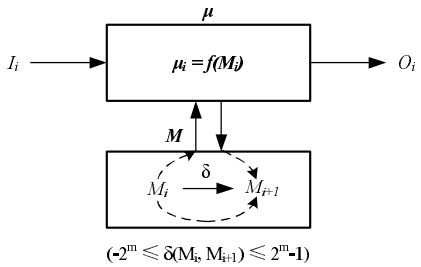


Fig. 2 m bits integer computing system state transformation

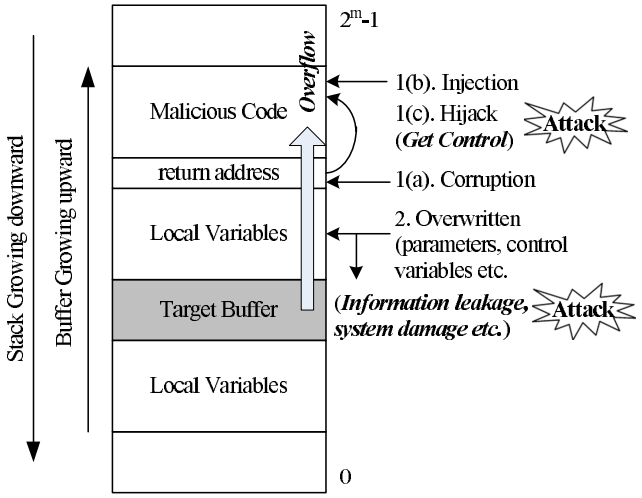


Fig. 3 Program runtime data layout

According to Eq. (1) and (2), it can propagate to pollute computing contexts and compromise the system eventually.

3.2 Buffer Overflow Attack Model

Taking data in processor Instruction Set Architecture(ISA) defined types as metadata, a buffer is a continuous block of homogeneous metadata items residing in program memory space. It can be defined as $BF_m(T_w, n)$ $\{w \in N+ \ and \ w \leq m/8, n \in N+\}$. T_w is a metadata type occupying w bytes memory and n is the buffer length. If $n = 1$, the buffer regresses to a scalar with programming language defined types(e.g. *char*, *short*, *int* in **C**), otherwise it is corresponding to an array. The k-th metadata can be referred as $BFD_m(T_w, k)$ $\{k \in N \ and \ k < n\}$.

Program runtime data layout in *little-endian* memory system is shown as Fig. 3. Typical attack scenarios are detailed in [27], [28]. The common concept is to extend $BF_m(T_w, n)$ to $BF_m(T_w, t)$ with $t > n$ and exploit program control with the additional $(t - n) * w$ bytes data. According to Sect. 3.1, polluted M_i affects O_i, M_{i+1} as well as μ_{i+1} and eventually attackers can hijack the program to achieve special purposes.

Figure 4 shows a typical vulnerable code and Fig. 5 illustrates its context after buffer overflow attack [29]. The possible attacks are:

(1) *Device attack*: Override key data such as device parameters to affect hardware running states and cause system damages.

(2) *Information Leakage*: Override control data such as branch conditions to affect program execution flow and provide a window for attackers to spy upon system secrets.

(3) *Control Hijack*: Override control data such as function return address to redirect program to injected malicious code and hijack the system.

Buffer overflow attacks succeed by feeding program with invalid structure of inputs to overflow program runtime contexts. Inputs with invalid values in valid data structures only trigger parameter errors instead of buffer overflows. Value validation of inputs is one of program's basic functionalities and not considered in this paper.

```
/* bf_attack.c */
int main(int argc, char ** argv) {
  int usr_priority;
  int motor_params[2];
  char input_text[64];

  scanf("%d", &motor_params[0]); //motor speed
  scanf("%d", &motor_params[1]); //motor voltage
  scanf("%d", &usr_priority);    //priority
  //!!!buffer overflow occurs here!!!
  scanf("%s", input_text);

  set_motor_speed(motor_params[0]);
  set_moter_voltage(motor_params[1]);
  if (usr_priority == 0) { //root debug
    print_motor_detail_status();
  } else {        //normal status print
    print_motor_speed();
  }

  return 0;
}
```

Fig. 4 Example code for buffer overflow attack

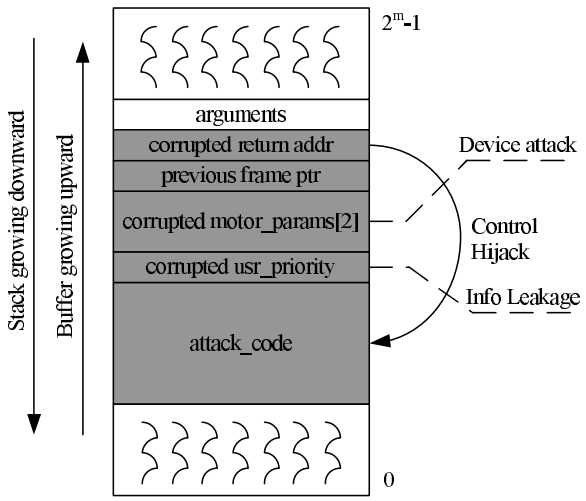


Fig. 5 Program context after buffer overflow attack

4. BFWindow Protection

4.1 Data Structure Coloring

For a 16 byte memory block, it can be interpreted as a 16 byte array or a 12 byte array plus 1 word as shown in Fig. 6. This "poly-interpretation" facilitates attacks with memory access crossing data structure boundaries.

To avoid such "poly-interpretation", data structure information should be recorded when memory is allocated. This paper takes data structure coloring method. By associating each byte with one additional property bit p, data structure view is clearly reflected with interleaving 0/1. Contigu-

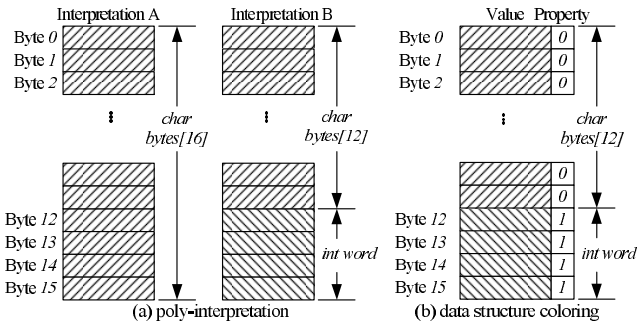


Fig. 6 Poly-interpretation and coloring of memory block

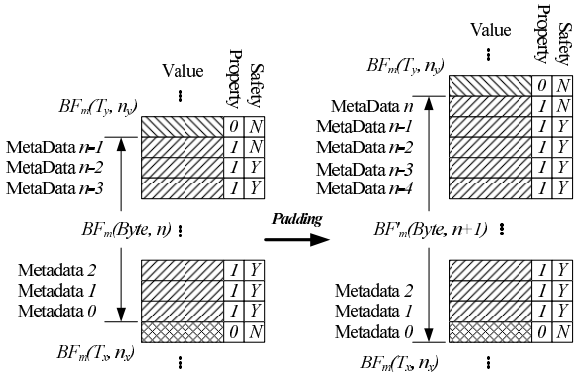


Fig. 7 Extending $BF_m(T_w, n)$ to $BF'_m(T_w, n+1)$ to avoid misinforming with *BFWindow(2)*

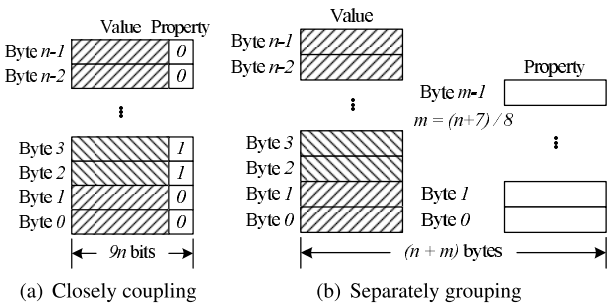


Fig. 8 Property bit map memory organization

ous bytes with same property belong to one data structure. And the boundary is drawn by property changes.

With data coloring mechanism, the data buffer definition in Sect. 3.2 is extended to $BF_m(T_w, n, p)$ $\{w \in N+ \text{ and } w \leq m/8, n \in N+, p \in \{0,1\}\}$. The k-th metadata item and its property are referred as $BFD_m(T_w, n, k)$ and $BFP_m(T_w, n, k)$ $\{k \in N \text{ and } k < n\}$ respectively.

4.2 Property Consistency Checking

Buffer overflow is triggered by multiple memory writes crossing data structure boundaries within the same program context. Instead of pairing corresponding writes for boundary checking which is impractical for hardware, this paper defines a buffer overflow free write based on speculatively data property consistency checking with following lemma.

Lemma 1: If a metadata has adjacent homogeneous metadata with the same data structure property in buffer growing direction, memory write to the metadata is safe.

Proof 1: The target and its adjacent metadata memory blocks can be defined as $M_m(addr, value, size, prop)$ and $M'_m(addr', value', size', prop')$.

$\because size' = size, addr' = addr + size, prop' = prop$
$\therefore \{M, M'\} = BF_m(size, 2, prop)$
$\therefore buffer\ boundary\ is\ not\ within\ BF_m(size, 2, prop)$
$\Longrightarrow write\ to\ BFD_m(size, 0, prop)\ safe$
$\Longrightarrow write\ to\ M_m(addr, value, size, prop)\ safe$

$BF_m(T_w, n, p)$ is also called *BFWindow(n)*. By extending target $BFD_m(T_w, k, p)$ to a corresponding *BFWindow(2)* and speculatively checking its property consistency, it's easy to validate legality of sequential writes.

For a buffer $BF_m(T_w, n, p)$ $(n \in N+, n \geq 2)$, *BFWindow(2)* convinces that writes to $\{BFD_m(T_w, k, n)$ $(k \in N, 0 \leq k \leq n-2)\}$ are always safe while the write to $BFD_m(T_w, n-1, n)$ is misinformed as unsafe. In order to get the correct set of *BFWindow*s, 1 additional homogeneous metadata is padded after $BF_m(T_w, n, p)$ to form a new $BF'_m(T_w, n+1, p)$ as Fig. 7. Writes to original buffer $\{BFD'_m(T_w, n, k)$ $(0 \leq k \leq n-1)\}$ are now safe while the write to the padded $BFD'_m(T_w, n+1, n)$ will be marked as unsafe. The padded item is used as embedded guard to trigger buffer overflow exception if sequential writes try to cross buffer boundary.

5. Implementation

5.1 Memory Organization

The property map needs to be stored for later consistency checking. It can be either closely coupled with each byte or grouped in a separate memory block as Fig. 8. Closely coupling method needs to modify current physical memory structure by extending 1 more bit per byte while separate memory block method can be easily implemented with memory space extending. This paper will take the separate bit map memory block for implementation and evaluation.

Property bits are stored in reserved memory which is unaddressable to normal memory access instructions at runtime. This defeats potential property overriden attacks.

5.2 Property Management

Property management includes property generation, initialization and cleanup. Enhanced compiler generates the property map during compilation. For stack buffers, compiler pads each buffer with one more homogeneous item and calculates properties of the new constructed buffer. It also automatically insert property initialization and cleanup code to function prologue and epilogue. For *BSS* and *data* buffers, padding items are also allocated during compilation. Initialization and cleanup code are inserted after *main*() entry and before *main*() exit respectively. While for heap buffers, enhanced *malloc*() and $free$() library functions perform guard

allocation, property initialization and cleanup. When an *array* is inside a *struct*, the *struct* is rewrited as Fig. 11 to maintain source compatibility.

Special instructions **SETP**(Set Property Bits) and **CLRP**(Clear Property Bits) are extended to ISA for property operation. Their syntaxes are shown as Fig. 9.

In order to optimize performance, this paper enables users to control buffer set to be protected. A basic idea is that input buffers exposed to attackers are unsafe while internal temporary buffers are safe. By removing unnecessary protections for safe buffers, it can minimize performance overhead. This paper introduces a "guided-protection" mechanism. The enhanced compiler supports different protection modes and a map file which contains the list of target buffers. The syntax of buffer protection map is shown in Fig. 10.

Combinations of protection modes and map files implement 3 levels of buffer protection as shown in Table 1.

We take CIL(C Intermediate Language) [32] and GCC to implement these features. The frontend tool automatically does all necessary instruments to source code which avoids extra efforts for application security upgrading.

```
Syntax : SETP/CLRP address, size
Example:
        char str[16];
        asm("SETP &str[0], sizeof(str)");
        asm("CLRP &str[0], sizeof(str)");
```

Fig. 9 SETP/CLRP instruction syntax

```
Syntax : file_name:func_name:var_name;
Example:
         //protect buffers in bf_attack.c
         bf_attack.c:main:motor_params;
         bf_attack.c:main:input_text;
```

Fig. 10 Buffer protection guide file syntax

```
struct bf {  // original definition
  func_ptr *ptr;
  char data[16];
} dat;

struct bf_ {  // rewrited definition
  func_ptr *ptr;
  char data_; //padding item
  char data[16];
} dat_;
memcpy(dat.data, dat_.data, sizeof(dat.data));
```

Fig. 11 Rewrite struct with buffers

Table 1 Protection level based on *BFWindow(2)*

Level	Mode	Map File	Targets
0	*–no-lbuf-prot*	Ignore	None
1	*–light-lbuf-prot*	Effective	Map file specified buffers
2	*–full-lbuf-prot*	Ignore	All local buffers

5.3 Property Checking

BFWindow(2) detects buffer overflow before memory being really accessed based on hardware. Take a classic five stage RISC processor pipeline for example, the checking is performed at "Memory Access" stage with "read-check-write" mechanism. Processor issues the data write transaction to memory bus. Memory controller reads corresponding property bits of target *BFWindow* from reserved property memory and checks the consistency. If the consistency is approved, it accepts the memory write and updates target memory block. Otherwise, buffer overflow is detected and it raise a memory bus error exception due to data structure property inconsistency. If data cache is presented, the same checking sequence is performed by cache controller instead.

As the memory write operation is after instruction commitment, the validation process dosen't block the instruction pipeline. Additionally, memory system performs the implicitly validation which eliminates additional checking instructions. These are important for performance sensitive embedded systems.

6. Results

This paper takes SimpleScalar [33] and GCC targeting ARM processor to validate security enhancement and evaluate performance overhead. *BFWindow(2)* protection and data structure map generation are implemented in SimpleScalar and CIL/GCC. Table 2 summarizes the target system model which is based on ARM SA-1100 core [34]. And the implementation source is available at *https://github.com/rynxr/BFWindow*.

Related protection methods introduced in Sect. 1 are evaluated on X86 desktop with age-old Ubuntu 6.06 which doesn't have any buffer overflow protection mechanisms and will not introduce interferences [35]. The practical combination of Address Space Layout Randomization(ASLR), NX and stack protector is evaluated in Ubuntu 12.04 with appropriate OS/compiler switches(*randomize_va_space* = 2, *-z noexecstack*, *-fstack-protector-all*). [12], [24]–[26] are

Table 2 System configurations for simulations

Architecture	Value
Instruction Fetch Queue	8 entries
Decode width	1 instruction per cycle
Issue width	1 instruction per cycle
Commit width	1 instruction per cycle
RUU(Register Update Unit)	4 entries
LSQ(Load/Store Queue)	4 entries
Function units	1 IALU, 1 IMUL, 1FPALU, 1FPMUL
Branch predictor	not taken
I/D-TLB	32-entry, 32-set, FIFO replacement, 30-cycle miss
L1 I/D-cache	16KB, 32-way, 32-byte cache line, FIFO replacement, 1-cycle hit
L2 I/D-cache	None
Memory system	1 port, pipelined, 64/1 cycle for first/inter chunk

Table 3 Buffer overflow protection efficiency comparison

Methods	Tech	Target Buffer Location				Sensitive Data Protection			Detection	R.T.	Failure
		stack	heap	BSS	data	ctrl_var	func_ptr	ret_addr			
No protection(Ubuntu6.06)	-	N	N	N	N	N	N	N	-	-	0%
Libsafe	Lib	Y/N	Y/N	Y/N	Y/N	Y/N	Y/N	Y/N	@Access	Hight	7%
StackShield	CC	Y	N	N	N	N	N	Y	@Refer	Low	36%
SRAS	HW	Y	N	N	N	N	N	Y	@Refer	Low	$\approx$ 36%
ProPolice	CC	Y	N	N	N	N	N	Y	@Refer	Low	40%
SecureBit	OS/HW	Y/N	Y/N	Y/N	Y/N	Y/N	Y	Y	@Refer	Low	$<$ 79%
CRED	CC	Y/N	Y/N	Y/N	Y/N	Y/N	Y/N	Y/N	@Access	High	79%
ASLR+NX+Stack-protector	CC/OS/HW	Y	Y	Y	Y	N	N	Y	@Refer	Low	90%
BFWindow(2)+no-lbuf-prot	-	N	N	N	N	N	N	N	-	-	0%
BFWindow(2)+light-lbuf-prot	CC/HW	Y	Y	Y	Y	Y	Y	Y	@Access	High	100%
BFWindow(2)+full-lbuf-prot	CC/HW	Y	Y	Y	Y	Y	Y	Y	@Access	High	100%

impractical to embedded system due to high memory overhead and not included for comparison.

6.1 Security Efficiency

The comprehensive buffer overflow dedicated testbed *RIPE* [36] is run for security validation. It crosses 5 attack dimensions including buffer locations(e.g. *stack*, *heap*, *BSS* etc.), target code pointers(e.g. *return address*, *function pointer* etc.), direct/indirect pointer overflow techniques, attack assist code classes(e.g. *shellcode*, *return-to-libc* etc.) and target vulnerable functions(e.g. *memcpy()*, *strcpy()*, *fscanf()* etc.) to provide 850 practical attack scenarios.

Table 3 shows results of security efficiency. In "Tech" column, *Lib/CC/HW/OS* denote that the implementations are based on library, compiler, hardware and operating system respectively. "Target Buffer Location" lists buffer locations which can be protected by corresponding methods. "Sensitive Data Protection" includes *ctrl_data*, *func_ptr* and *ret_addr* which are abbreviations of control data(e.g. branch condition, pointer table index etc.), function pointer and function return address. "Detection" column indicates when attacks are identified, either at *data access* stage or *data referring* stage. "R.T." column summarize the attack response timeliness level corresponding to "Detection". "Failure" column lists the experiment results of applying *RIPE*.

Result "Y" means that the method supports protecting targets and "N" means not. "Y/N" means corresponding method either effective or ineffective for different programs. For example, Libsafe can only protect buffers operated by enhanced library functions, ProPolice can't protect adjacent buffers in buffer growing direction and CRED targets string buffers only to reduce performance overhead. Though SRAS and SecureBit are hardware methods which cannot be simulated, SRAS is similar to StackShield and SecureBit only targets control data flowing to Program Counter(PC) which is weaker than CRED, their "Failure" are derived from StackShied and CRED respectively.

Most protections like StackShield, SRAS, ProPolice, SecureBit and NX detect attacks at data referring stage such as function returns to corrupted address. It may be a long period after the attack occurrence. Libsafe and CRED perform boundary checking at data access stage. However, home-brew code and performance issues limit their practicability. *BFWindow(2)* aggressively validates sequential memory writes based on underlying data layout. It timely prevents malicious inputs from polluting program runtime contexts.

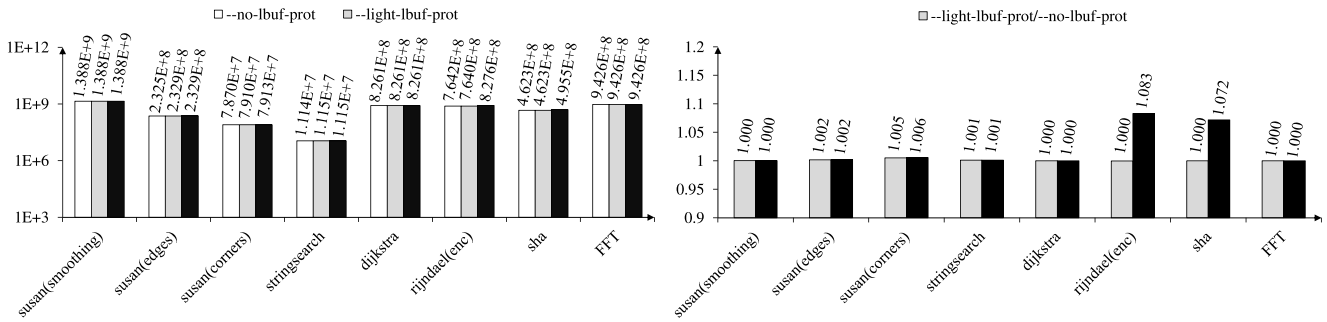
Exploitations from real applications are also evaluated. CVE-2015-0235("GHOST") is from *glibc* which overflows caller-supplied buffers(*heap*, *stack*, *bss etc.*). And cve-2015-5291 from *mbed TLS* overflows heap buffers. Both have serious impacts on information security of embedded systems. *BFWindow(2)* can effectively detect attacks as soon as memory writes crossing buffer boundaries[†].

6.2 Performance Overhead

Embedded MiBench [37] is adopted for performance evaluation with following selected benchmarks: susan - an image package; string search - search algorithm for given words in phrases; dijkstra - an Dijkstra's algorithm implementation; FFT - Fast Fourier transforms used in digital signal processing; sha - the standard secure hashing algorithm; rijndael - a popular symmetric cryptography algorithm used in industry. All benchmarks are simulated with large data set inputs. Taking system without buffer protection mechanism as baseline, Fig. 12(a) shows the total simulation cycles and Fig. 12(b) shows ratios of normalized simulation time.

The performance overhead mainly comes from data structure property map management and *struct* rewrites. It's highly depends on the protected buffer scale of programs. The full protection mode protects all buffers compulsively. It introduces less then 1% performance overhead for most benchmarks except 8.3% for rijndael and 7.2% for sha. The large overheads are due to that these 2 algorithm implementations uses large number temporary buffers which introduces unnecessary protections. For example, rijndael takes 2 local buffers, each with 16 bytes, as state memory for each round and large lookup tables for substitution and multiplication on $GF(2^8)$, while sha aggressively allocates 8192 bytes to temporarily hold data slices. In light protection mode simulation, it protects buffers related to inputs only and filters out the internal safe buffers. The performance penalty for light mode protection are under 1%.

[†]https://github.com/rynxr/BFWindow/tree/master/src/cve



(a) Execution cycles (b) Normalized execution time

Fig. 12 Performance results with different *BFWindow(2)* protection levels

Hardware resource overhead contains instruction extension logic and property map memory space. Both **SETP/CLRP** are similar with normal memory instructions(e.g.*store*) and the extra decoding logics are negligible. The memory overhead includes property bits and padding items. As a property bit is associated to each byte, it takes extra 12.5% memory in total. But it can dramatically decreases the overhead to 3.125% by padding buffers, aligning them with 4 bytes boundary and associating property bits to words which are practical for common 32 bit systems. System designers can also remove unnecessary property bits for scalars. By separating buffer memory space with properties to scalar memory space without properties, it further decreases property memory overhead. And the space of padding items depends on the number of protected buffers. It is acceptable for security system designers considering the gained security enhancements.

7. Conclusion

This paper proposed a technique called *BFWindow* to protect program buffers from buffer overflow attacks caused by sequential memory writes. Based on data structure coloring with single property bit and speculatively property consistency checking of target *BFwindow*(2), it aggressively prevents unsafe memory writes at early data access stage. The malicious inputs with invalid data structures can never cross a buffer boundary to pollute other local data. The security enhancements are proved by both theory and experiments. Property bits reside in an unaddressable memory space which also prevents potential overridden attacks.

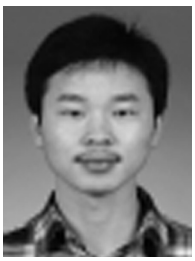
We also implement guided protection mechanism in compiler with fine control of protection targets to minimize performance overhead. Performance evaluation results show that the practical light protection of input buffers introduces less then 1% performance loss. All instruments are transparent to programmers and it's easy to deploy for security upgrading without extra efforts or risks. Hardware based checker resides in memory system minimizes impacts on processor design and runtime performance.

By extending *BFWindow(2)* to *BFWindow(3)* with two separate guard items before/after target buffer, it can effectively protect buffers from overflow as well as underflow attacks. Furthermore, enhancing $free()$ to scramble property bits of freed memory block with interleaving 0/1, it prevents accidently access to unallocated memory which is know as "use-after-free" and "double-free" attacks of pointers.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," Comput. Netw., vol.54, no.15, pp.2787–2805, 2010.
[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems, vol.29, no.7, pp.1645–1660, 2013.
[3] H. Li, M. Dong, and K. Ota,, "Radio Access Network Virtualization for the Social Internet of Things," IEEE Cloud Computing, vol.2, no.6, pp.42–50, 2015.
[4] J. Wu, M. Dong, K. Ota, L. Liang, and Z. Zhou, "Securing distributed storage for Social Internet of Things using regenerating code and Blom key agreement," Peer-to-Peer Networking and Applications, vol.8, no.6 pp.1133–1142, 2015.
[5] S.K. Sowe, T. Kimata and M. Dong, and K. Zettsu, "Managing Heterogeneous Sensor Data on a Big Data Platform: IoT Services for Data-Intensive Science," COMPSAC Workshops, pp.295–300, 2014.
[6] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, "Security as a new dimension in embedded system design," Proc. 41st annual Design Automation Conference, pp.753–760, 2004.
[7] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," ACM Trans. Embedded Computing Systems (TECS), vol.3, no.3, pp.461–491, 2004.
[8] M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace and kernelspace," USENIX Security Symposium, pp.395–410, 2008.
[9] CERT, OpenSSL heartbleed, https://www.us-cert.gov/ncas/alerts/TA14-098A, 2014.
[10] G.C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," ACM SIGPLAN Notices, vol.37, no.1, pp.128–139, 2002.
[11] O. Ruwase and M.S. Lam, "A practical dynamic buffer overflow detector," NDSS, pp.159–169, 2004.
[12] P. Akritidis, C. Cadar, C. Raiciu, and M. Castro, "Preventing memory error exploits with WIT," IEEE Symposium on Security and Privacy, pp.263–277, 2008.
[13] C. Cowan, C. Pu and D. Maier, et al., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," USENIX Security Symposium, vol.98, pp.63–78, 1998.
[14] S.S. Vendicator, "A stack smashing technique protection tool for linux," http://www.angelfire.com/sk/stackshield/info.html
[15] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," Recent

Advances in Intrusion Detection, pp.121–141, Springer, 2011.
[16] H. Etoh and K. Yoda, "Propolice: Improved stack-smashing attack detection," IPSJ SIGNotes Computer Security, 2001.
[17] NX bit, https://en.wikipedia.org/wiki/NX_bit
[18] J.P. McGregor, D.K. Karig, Z. Shi, and R.B. Lee, "A processor architecture defense against buffer overflow attacks," Information Technology: Research and Education, pp.243–250, 2003.
[19] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," OSDI, vol.8, pp.225–240, 2008.
[20] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B.C. Pierce, H. Reubenstein, J.M. Smith, G.T. Sullivan, A. Thomas, J. Tov, C.M. White, and D. Wittenberg, "SAFE: A clean-slate architecture for secure systems," IEEE International Conference on Technologies for Homeland Security (HST), pp.570–576, 2013.
[21] J.R. Crandall and F.T. Chong, "Minos: Control data attack prevention orthogonal to memory model," MICRO-37, pp.221–232, 2004.
[22] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," ACM SIGPLAN Notices, vol.39, no.11, pp.85, 2004.
[23] K. Piromsopa and R.J. Enbody, "Secure bit: Transparent, hardware buffer-overflow protection," IEEE Trans. Dependable and Secure Computing, vol.3, no.4, pp.365–376, 2006.
[24] S. Nagarakatte, M.M.K. Martin, and S. Zdancewic, "Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety," 39th Annual International Symposium on Computer Architecture, 2012.
[25] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," 319433-015 edition, 2013.
[26] S. Nagarakatte, M.M.K. Martin, and S. Zdancewic, "WatchdogLite: Hardware-accelerated compiler-based pointer checking," International Symposium on Code Generation and Optimization, pp175, ACM, 2014.
[27] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," DARPA Information Survivability Conference and Exposition, vol.2, pp.119–129, 2000.
[28] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," World Wide Web, 2002.
[29] Z. Shao, Q. Zhuge, Y. He, and E.H.-M. Sha, "Defending embedded systems against buffer overflow via hardware/software," IEEE Computer Security Applications Conference, pp.352–361, 2003.
[30] S. Jisha, D. Thomas, and S. Jamal, "A categorized survey on buffer overflow countermeasures," IJARCCE, vol.2, pp2068–2074, 2013.
[31] A. Baratloo, N. Singh and T.K. Tsai, et al., "Transparent run-time defense against stack-smashing attacks," USENIX Annual Technical Conference, General Track, pp.251–262, 2000.
[32] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of c programs," Compiler Construction, pp.213–228, Springer, 2002.
[33] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," Computer, vol.35, no.2, pp.59–67, 2002.
[34] Intel, "SA-1100 Microprocessor Technical Reference Manual," 1998.
[35] Canonical, Ubuntu Wiki, https://wiki.ubuntu.com/Security/Features, 2016.
[36] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: runtime intrusion prevention evaluator," Proc. 27th Annual Computer Security Applications Conference, pp.41–50, ACM, 2011.
[37] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," Workshop on Workload Characterization, pp.3–14, IEEE, 2001.



Jinli Rao was born in 1987. He received the B.S. and M.S. degrees in Microelectronics and Solid State Electronics from Huazhong University of Science and Technology, China in 2009 and 2012 respectively. Currently He is a Ph.D candidate in Huazhong University of Science and Technology, China. His research interests include information security, computer architecture and VLSI design.

Zhangqing He received his B.S. Degree from Hubei University of Technology and M.S. Degree from Huazhong University of Science and Technology, China in 2003 and 2008. Currently he is a Ph.D student in Huazhong University of Science and Technology and also is a associate professor in Hubei University of Technology, China. His research interests include embedded system and security.

Shu Xu received B.S. degree in mathematics from Peking university in 1984 and Ph.D. degree in cryptology from Zhengzhou Information Science and Technology Institute in 2008 respectively. Now he is a research fellow in Science and Technology on Information Assurance Laboratory. His research interests include computer architecture, cryptology and information security. He has been awarded 1 time the special grade prize, 1 time the first prize and 2 times the second prize of national award for Science and Technology Progress, 7 times the first prize of ministerial awards, 7 times the second prize of ministerial awards.

Kui Dai received B.S. degree in information science and technology from Harbin University of Technology and the Ph.D. degrees in Computer Science and Technology from National University of Defense Technology, China in 1989 and 1994 respectively. He joined the National University of Defense Technology in 1995. Since 2008 he was a professor in Huazhong University of Science and Technology. Now he is a professor in the Institute of National Network Security and Information, Peking University.

Xuecheng Zou received the B.S. M.S. and Ph.D. degrees, all in electrical engineering, from Huazhong Unversity of Science and Technology, China in 1985, 1988 and 1995 respectively. He is currently a professor and doctoral supervisor in School of Optical and Electronic Information, Huazhong University of Science and Technology. His research interests span microelectronics and solid state electronics, information security, VLSI design.