SafeStack$^+$: Enhanced Dual Stack to Combat Data-Flow Hijacking

Yan Lin, Xiaoxiao Tang, and Debin Gao

School of Information Systems, Singapore Management University, Singapore
{yanlin.2016, xxtang.2013, dbgao}@smu.edu.sg

Abstract. SafeStack, initially proposed as a key component of Code Pointer Integrity (CPI), separates the program stack into two distinct regions to provide a safe region for sensitive code pointers. SafeStack can prevent buffer overflow attacks that overwrite sensitive code pointers, e.g., return addresses, to hijack control flow of the program, and has been incorporated into the Clang project of LLVM as a C-based language front-end. In this paper, we propose and implement SafeStack$^+$, an enhanced dual stack LLVM plug-in that further protects programs from *data-flow* hijacking. SafeStack$^+$ locates data flow sensitive variables on the unsafe stack that could potentially affect evaluation of branching conditions, and adds canaries of random sizes and values to them to detect malicious overwriting. We implement SafeStack$^+$ as a plug-in on LLVM 3.8 and perform extensive experiments to justify a lazy checking mechanism that adds on average 3.0% of runtime and 5.3% of memory overhead on top of SafeStack on SPEC CPU2006 benchmark programs. Our security analysis confirms that SafeStack$^+$ is effective in detecting *data-flow* hijacking attacks.

Keywords: buffer overflow, data flow, control flow

1 Introduction

Many techniques have been proposed to fight against memory attacks, e.g., Data Execution Prevention (DEP) [8] to prevent code execution in non-executable memory regions, Address Space Layout Randomization (ASLR) [3] to randomize the location where executable is loaded into memory, Control Flow Integrity (CFI) [31, 26, 23] to prevent redirecting of execution flows. Code Pointer Integrity (CPI) [22] is a recent addition to the family of defenses to provide integrity of code pointers in a program and thereby prevent control-flow hijacking attacks. The core of CPI is a C-based language front-end of LLVM called *SafeStack* that splits the regular stack into two parts: a safe stack and an unsafe stack. All proven-safe objects are placed onto the safe stack while those that cannot be proven safe are placed onto the unsafe stack, such as buffers which may overflow. SafeStack prevents a buffer overflow on the unsafe stack from corrupting anything on the safe stack, and thereby prevents control-flow hijacking attacks.

It introduces negligible runtime overhead of less than 0.1%, and has been incorporated into the Clang project of LLVM due to its increasing acceptance by developers.

Being proposed as a defense to provide code pointer integrity, SafeStack, however, is susceptible to data-flow hijacking attacks. In particular, objects on the unsafe stack could overwrite each other, and such unsafe objects could potentially be used subsequently in an evaluation of branch conditions, changing which would lead to a successful data-flow hijacking attack. In this paper, we propose SafeStack$^+$, an enhanced dual stack mechanism that works on top of SafeStack to detect data-flow hijacking attacks. The idea of SafeStack$^+$ is to add protections into the unsafe stack rather than leaving it as the attackers' playground. SafeStack$^+$ first locates all variables on the unsafe stack that could potentially affect the execution of conditional branches using a def-use analysis, and then adds canaries of random sizes and values around them. Finally, SafeStack$^+$ adds runtime checks into the program to verify the integrity of the canaries to detect data-flow hijacking attacks.

Although the idea sounds simple, the key to a successful defense of memory attacks that can gain acceptance by developers is a low runtime overhead in the resulting binary executable. To achieve this goal, we implement SafeStack$^+$ on LLVM 3.8 with various canary checking mechanisms to test the corresponding runtime overheads. The extensive experiments show that our *lazy checking* mechanism that verifies the integrity of canaries at the point of branching evaluation results in a small runtime overhead of 3.0% and memory overhead of 5.3% on average on top of SafeStack. We further confirm SafeStack$^+$'s enhanced security with a real-world vulnerability CVE-2013-0230.

In summary, this paper makes the following contributions:

1. We propose SafeStack$^+$, an LLVM plug-in on top of SafeStack that adds canaries around sensitive objects on the unsafe stack to detect data-flow hijacking attacks.
2. We perform extensive testing on various canary checking mechanisms to justify our lazy checking technique, and show that it results in low runtime and memory overhead.
3. We demonstrate that SafeStack$^+$ can be used to effectively defend against data-flow hijacking attacks with a real-world vulnerability.

The remainder of this paper is structured as follows. We first discuss in Section 2 the limitation of SafeStack and our motivation. Section 3 introduces the design and implementation of SafeStack$^+$. We demonstrate the efficiency of SafeStack$^+$ with extensive performance evaluations and present the security analysis in Section 4. Section 5 briefly introduces the related work on memory corruption countermeasures and points out the limitation of SafeStack$^+$. In the end, we conclude in Section 6.

2 SafeStack and Our Motivation

As mentioned in Section 1, SafeStack is a core component of Code Pointer Integrity (CPI). In this section, we first briefly discuss how SafeStack works and its limitations. After that, we present our motivation of SafeStack$^+$ in tackling SafeStack's limitations.

2.1 SafeStack

Kuznestsov et al. [22] proposed Code Pointer Integrity (CPI) to guarantee the integrity of all code pointers in a program (e.g., function pointers and saved return addresses) by storing the sensitive pointers and their metadata (which describes the target object on which the sensitive pointer is based) in a safe memory region. Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe using the metadata associated with the pointer being dereferenced. CPI treats the stack specially, because the safety of most accesses to stack objects requires no runtime checks as they can be checked statically during compilation.

SafeStack is used to protect critical data on the stack by separating the native stack into two areas. There is a safe stack which is used for control flow information and data that is only ever accessed in a safe way (safe in the sense that the pointer dereference is safe – the memory it accessed lies within the target object on which the dereferenced pointer is based). There is an unsafe stack which is used for everything else that is stored on the stack. By arranging information on the two separated stacks, the safe stack can be accessed without any checks. The two stacks are located in different memory regions in the process's address space and thus prevents a buffer overflow on the unsafe stack from corrupting anything on the safe stack.

Listing 1 shows an example where we indicate results of the static analysis in SafeStack as comments below the code lines. We encourage readers to refer to the original paper of CPI [22] and source code of SafeStack [2] for the precise definitions.

SafeStack is implemented as a plug-in of LLVM to statically analyze source code of a program to identify its safe and unsafe objects. After identifying the safe and unsafe objects, SafeStack allocates space for unsafe objects in the unsafe memory region, which is accessible through a dedicated segment register (`%gs` in x86-32). Unsafe objects are placed onto the unsafe stack next to each other to minimize memory overhead.

Although the design of SafeStack meets the requirement of minimal memory overhead which is likely an important reason why it has been gaining developers' acceptance, it leads to an important limitation – unsafe objects are located at predictable locations on the unsafe stack, and could overwrite one another in a predictable manner.

Figure 1 shows the layout of the safe and unsafe stacks when the code in Listing 1 executes. We notice that all unsafe objects are pushed onto the unsafe stack one next to the other, and other proven-safe objects are stored on the

Listing 1. Example code

```
1  void determine_privilege_level(int *pl) {
2      *pl = get_priviliege();
3  }
4  int main() {
5      int i = 0;
6      int pl;
7      int *ptr;
8      char buffer[16];
9      int p[16];
10     int b = 1;
11     int len;
12
13     ptr = &b;
14     /*b is unsafe -- conservatively assume that storing a pointer is unsafe as
15     there's no way to tell whether it points to a valid object or not.*/
16
17     memset(p,0,20);
18     /*p is unsafe -- the size of memory access region of p is 20, greater than
19     the allocated size of 16.*/
20
21     determine_privilege_level(&pl);
22     /*pl is unsafe -- potential information leak when a pointer to a local
23     variable is passed to another function.*/
24
25     gets(buffer);
26     /*buffer is unsafe -- potential information leak when a pointer to a local
27     variable is passed to another function.*/
28
29     len = pl;
30     if (pl == 0x42)
31        access_file(FILE *f);
32     else
33        printf("Not allowed to access the file");
34
35     return &i;
36     /* i is unsafe -- returning a pointer may cause information leakage.*/
37 }
```

safe stack. This ensures that unsafe objects could not modify objects on the safe stack; however, e.g., `pl` can be overwritten by `buffer` and `p` in a typical buffer overflow, and since the offsets between `pl` and the buffers are fixed and can be easily learned from the code, the overwriting of `pl` is easy and its effect is predictable by an attacker.

To make things worse, `pl` is a *sensitive* variable in the sense that its value determines the branching decision at line 30 of Listing 1, which makes the overwriting of `pl` a successful data-flow hijacking attack.

2.2 Motivation

As shown in Section 2.1, SafeStack defeats control-flow hijacking attacks with minimal overhead, but it is vulnerable to data-flow hijacking attacks. This limitation stems from the fact that SafeStack was initially proposed in the project of

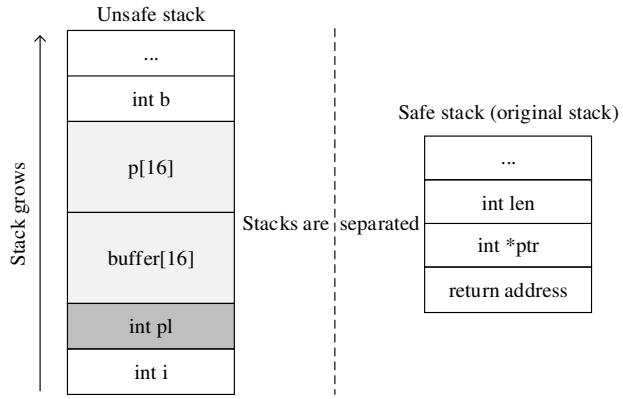

Fig. 1. Layout of the unsafe stack and safe stack

Code Pointer Integrity (CPI) which concerns only control flow integrity. In this paper, we investigate the possibility of enhancing SafeStack such that it is resistant to data-flow hijacking attacks while retaining the advantages of SafeStack in its negligible runtime overhead.

A simple yet effective way of protecting data-flow sensitive variables is to add canaries around them and to check for integrity of the canaries during program execution. However, in such an approach, it is crucial to precisely identify sensitive data for canary introduction to minimize the potential overheads – runtime overhead for checking their integrity and memory overhead for storing the canaries. Not only that, the runtime checking of the canaries also need to be efficient enough not to cause excessive runtime overheads. In this paper, we focus on protecting data whose values could potentially affect the evaluation of conditional branches (e.g., `pl` as in Listing 1; referred to as sensitive data in the rest of the paper), since other control-sensitive data (e.g., the return addresses) are already well protected in the safe stack.

3 Design and Implementation of SafeStack$^+$

To show how SafeStack$^+$ achieves its objects in defending against data-flow hijacking attacks, we present our design and implementation of SafeStack$^+$, an enhanced dual stack mechanism built on top of SafeStack. SafeStack$^+$ is an enhancement to SafeStack in the sense that it retains the dual-stack design of SafeStack and its definition of safe and unsafe objects. SafeStack$^+$ achieves its design objective by introducing canaries in the unsafe stack to detect modifications to sensitive data.

In this section, we will begin with the threat model of SafeStack$^+$, and then present its detailed design. We then present some implementation details of SafeStack$^+$ to improve its performance.

3.1 Threat Model

This paper is concerned with control flow (return) and data flow hijacking attacks, namely ones that give the attacker control of the return targets and ones in which the attacker can overwrite data that affect the execution of conditional branches. The purpose of the former type of attacks is to divert control flow to a location that would not otherwise be reachable in the same context, whereas the latter is to corrupt the decision making data and make the program execute another path.

We assume that the attacker can fully control over the process memory, but he does not have the ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segments as code pages are marked read-executable and not writable. Meanwhile, they cannot control the program loading process. These assumptions ensure the integrity of the original program code instrumented at compile time, and enable the program loader to safely configure the dedicated segment register used by the canaries and the unsafe stack.

3.2 Design

The high-level design of SafeStack$^+$ follows that of SafeStack in that both consist of a static analysis pass that identifies important objects in a program P (sensitive unsafe variables in the case of SafeStack$^+$) and an instrumentation pass that rewrites P to protect the important objects. However, SafeStack$^+$ differs from SafeStack in that we introduce canaries to further protect unsafe objects on the unsafe stack. In this section, we present our detailed design of SafeStack$^+$.

Figure 2 shows the workflow of SafeStack$^+$. We run static analysis to find sensitive variables first, and then instrument the code for canary insertion and runtime canary checking.

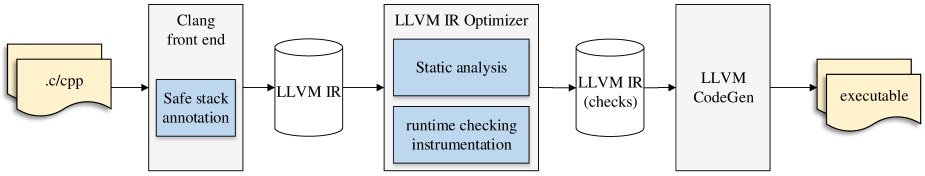

Fig. 2. Illustration of SafeStack$^+$'s workflow

Static Analysis We determine the set of sensitive variables using def-use analysis in LLVM where a variable is considered sensitive if it affects the execution of conditional branches and it is unsafe. The definition of an unsafe variable follows that in SafeStack that the pointer dereference of it is unsafe – the memory it

accessed may not lie within the target object on which the dereferenced pointer is based. We could do this via a forward execution analysis by keeping track of all code locations where an unsafe variable is used (directly and indirectly). If the target location involves a conditional branch, we add the unsafe variable to the sensitive set. Alternatively, we can also perform a backward tracking analysis starting from the sink of conditional branches, and trace back to the unsafe variables as sources. Since we implement SafeStack$^+$ as an LLVM plug-in and perform the static analysis during compilation time, we choose the former method for its simplicity. Note that if an unsafe variable is used as the argument of a call instruction, the analysis needs to jump inside the callee function to check whether the arguments will be used by a conditional branch.

Instrumentation for Runtime Checking We protect the sensitive unsafe variables by inserting canaries around them and checking the integrity of the canaries at runtime. A canary is a piece of data inserted on the stack to detect memory corruption attacks [13]. For example, if any buffer is overflown in an attack, the canary on the stack is likely overwritten before the sensitive data next to it is modified. Therefore, checking the integrity of the canaries enables detection of memory corruptions. Both inserting and checking the canaries are done via instrumenting the target program during compilation.

As shown in Figure 3, a canary is added next to the sensitive variable at the lower address (toward the direction of stack growth) to detect overwriting by other unsafe variables from lower addresses. To deal with brute-force attacks, canaries added in SafeStack$^+$ do not have fixed sizes or values. There is a trade-off between security and performance when setting the maximum size of the canaries – bigger size gives better security in that it provides higher entropy to the canary value, but also adds more runtime overhead to checking its integrity and bigger memory usage. SafeStack$^+$ randomly chooses from three different sizes: 4, 8, and 16 bytes (for memory alignment purposes) at compile time. Canaries are accessible through a dedicated segment register (`%gs`) to prevent attackers from obtaining them easily.

After canaries are inserted next to the sensitive variables, we need to check its integrity at runtime. The time of checking also involves trade-off between security and performance: checking integrity at every access of the sensitive variable (reading from and writing to) gives better security, but the frequent access might introduce prohibitive overhead. We introduce a lazy checking mechanism by delaying the integrity check till the point of conditional branch evaluation. We consider this an acceptable security policy since SafeStack$^+$ is designed to fight against data-flow hijacking attacks, and the lazy checking right before branching satisfies the security requirement. Although it may lead to a delay in detecting the corresponding attack, it could greatly improve performance due to the lower checking frequency.

Figure 4 shows the different locations to check the integrity of the canary for the sensitive variable `p1` in the sample code in Listing 1. We can perform checking

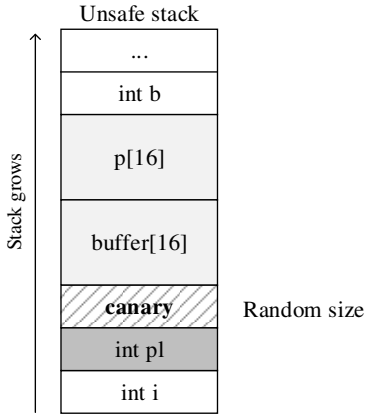

Fig. 3. SafeStack$^+$ Approach

– When reading the value of the sensitive variable from memory, see Figure 4(a);
– When storing the value of the sensitive variable to memory, see Figure 4(b);
– or, when evaluating the branching condition, see Figure 4(c).

```
void determine_privilege_level(int *p){
    *pl = get_priviliege();
}
int main(){
    ……
    determine_privilege_level(&pl);
    gets(buffer);
    // canary checking
    len = pl;
  // canary checking
    if ( pl == 0x42 )
        printf("Accessed\n");
    else
        printf("Not allowed to access the file\n");
    return 0;
}
```

(a) When reading from memory,

```
void determine_privilege_level(int *p){
    *pl = get_priviliege();
    // canary checking
}
int main(){
    ……
    determine_privilege_level(&pl);
    gets(buffer);
    len = pl;
    if ( pl == 0x42 )
        printf("Accessed\n");
    else
        printf("Not allowed to access the file\n");
    return 0;
}
```

(b) When storing to memory,

```
void determine_privilege_level(int *p){
    *pl = get_priviliege();
}
int main(){
    ……
    determine_privilege_level(&pl);
    gets(buffer);
    len = pl;
    // canary checking
    if ( pl == 0x42 )
        printf("Accessed\n");
    else
        printf("Not allowed to access the file\n");
    return 0;
}
```

(c) When evaluating branch conditions,

Fig. 4. Lazy checking of integrity of canaries

Intuitively, delaying the checking at branching evaluation might result in the smallest number of checks because each branching evaluation might correspond to multiple reads and writes of the sensitive variable. We delay our further discussion on the design choice to Section 4 where we discuss the experiments performed, since we want to measure the amount of saving before making the design decision.

3.3 Implementation

We obtained the source code of SafeStack integrated in LLVM 3.8 compiler infrastructure [5], and added more than 700 lines of code in C++ to implement SafeStack$^+$. Most of the additional code is added to perform static analysis to find instructions that manipulate the sensitive variables and to add instructions to check the value of the canary. Some code is also added to add canaries when allocating space for these sensitive variables. SafeStack$^+$ accepts unmodified `C/C++` program source as its input.

Sensitive Variable Analysis We implement the sensitive data analysis for SafeStack$^+$ as an LLVM pass. The LLVM pass operates on the LLVM Intermediate Representation (IR), which is a low-level strongly-typed language-independent program representation tailored for static analysis and optimization. The LLVM IR is generated from the `C/C++` source code by clang [1], which preserves most of the type information that is required by our analysis and the def-use chain can be used easily to get the locations for each variable.

For every unsafe `alloca` instruction that allocates memory on the stack frame, we traverse the list of instructions that make use of it using the def-use chain provided by LLVM. If the corresponding memory is used by a conditional branch, we consider it as sensitive. When checking whether arguments of each function call are involved in the evaluation of a conditional branch, if the caller and callee are located in different modules, it will be difficult to carry out the analysis. We take a simple solution to first compile all source code into one IR file.

Note that the determination of sensitive variables is a conservative process – a sensitive variable may not be overwritten forever as there are no vulnerable buffers being stored beyond it. We leave a more precise static analysis to find sensitive variables our future work.

Canary Insertion and Integrity Check Canaries on SafeStack$^+$ are stored in the thread control block which can be accessed only directly through one of the segment registers. We implement this by using the `InitialExecTLSModel` flag. To insert a canary to protect a sensitive variable, we modify function `moveStaticAllocasToUnsafeStack()` so that a canary is created right after allocating spaces for unsafe variables on the unsafe stack.

Integrity checking of the canaries at every reading (or writing) of the sensitive variables is implemented by traversing the list of instructions that make use of the corresponding sensitive variable, and inserting a call before (or after) the instruction to check for integrity. Our lazy checking, on the other hand, is implemented by finding conditional branches whose evaluation is affected by sensitive variables and inserting a call before the evaluation to check for integrity of the Canaries. We do not make additional effort to optimize the instrumentation code (e.g., by inlining the code of integrity checking instead of inserting a function call) because the compiler will perform further compilation and optimization after our instrumentation.

4 Evaluation

In this section, we perform a number of experiments to demonstrate the efficiency and effectiveness of SafeStack$^+$. Specifically, we first perform some simple statistical analysis on software programs to find out the number of variables that require protection in order to defend against data-flow hijacking. After that, we empirically test a number of ways of implementing our idea to justify our lazy checking mechanism. Finally, we test SafeStack$^+$'s capability in defending against a suite of security attacks and a real-world data-flow hijacking attack.

All experiments were performed on a desktop computer with an Intel i7 4510u CPU with 8GB of memory running the x86 version of Ubuntu 14.04. All experiments were conducted 10 times, average of which is reported in this paper.

4.1 Variables to be Protected for Data-flow Hijacking

The first experiment we performed is to find out how many sensitive variables need to be protected to defend against data-flow hijacking. If there are many, then it may make sense to just add canaries for every one and skip the process of locating sensitive ones. Table 1 shows some simple statistics for SPEC CPU2006 programs compiled without optimization. Specifically, we show percentage of unsafe functions (functions with at least one unsafe variable) upon all functions, percentage of unsafe variables upon all variables, and percentage of sensitive variables upon all unsafe variables.

Table 1. Simple statistics of SPEC benchmark programs

Program	Unsafe Functions	Unsafe Variables	Sensitive Variables
`bzip2`	23.3%	8.3%	44.4%
`gcc`	13.1%	4.0%	47.0%
`mcf`	8.3%	4.4%	87.5%
`sjeng`	27.1%	13.2%	60.4%
`libquantum`	33.0%	8.6%	14.9%
`astar`	15.1%	9.7%	21.3%
`namd`	43.2%	4.6%	78.9%
`soplex`	11.9%	7.7%	19.5%
`lbm`	28.5%	8.6%	18.2%
`average`	22.6%	7.7%	43.6%

The first two columns of results basically show that there are not that many functions requiring an unsafe stack, and there are not that many unsafe variables on the unsafe stack when it is needed. This explains why, in general, SafeStack has small overheads. The last column of results, which are more specifically about SafeStack$^+$, show that the percentage of sensitive variables upon all unsafe variables covers a relatively big range from 18% to 87%. That said, the average

is still below 50%, which justifies our strategy of locating only sensitive variables for added protection.

Note that the analysis above is purely static, which may not closely correspond to the overhead experienced by end users. We therefore need some dynamic analysis in order to precisely find out the user experience in terms of runtime overhead.

4.2 Dynamic Analysis for Various Strategies of Integrity Check

Having shown that there are fewer than 50% of the unsafe variables requiring protection against data-flow hijacking attacks, we now move on to dynamically analyzing the overhead when the benchmarking programs are running on certain workloads. At the same time, we also want to try out different integrity checking mechanisms to test the extent to which our intuition of lazy checking generating less overhead is correct. Figure 5 shows the results for our three canary integrity checking strategies – before reading sensitive variables from memory, after storing them to memory, and before evaluating branching conditions. Please refer to Section 3.2 for more discussions of the three strategies. Note that here we show the additional overhead of dynamically executing the benchmarking programs on SafeStack$^+$ over that on SafeStack, when the programs are given the workload of the largest input file under the `ref` folder provided by SPEC CPU2006.

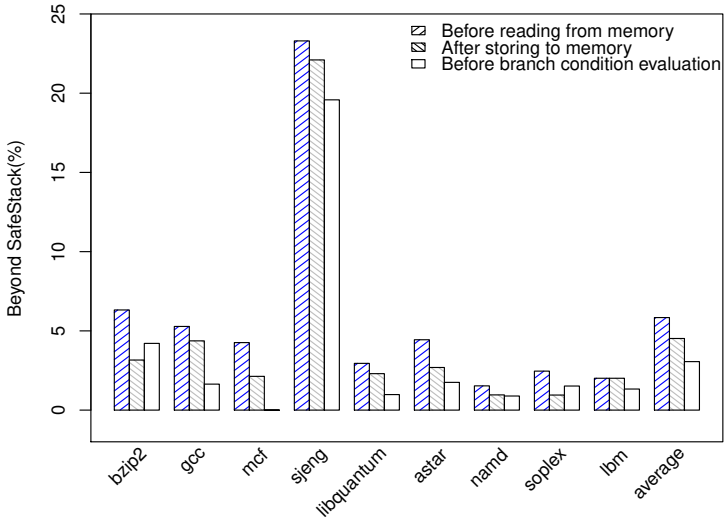

Fig. 5. Additional runtime overhead of SafeStack$^+$ over SafeStack

Results show that performing integrity check before branch condition evaluation enjoys a smaller runtime overhead of 3.0% compared to 5.8% and 4.5%, respectively, when checking before reading or after storing the sensitive variables. This confirms our intuition since each branch condition evaluation may correspond to multiple variable reads and writes. We therefore decide that SafeStack$^+$ shall adapt the lazy checking mechanism for improved efficiency. `sjeng` experiences much higher runtime overhead than other programs. We investigate the detailed execution, and find that this is due to a large number of looping that result in more integrity checking needed. That said, our general finding of lazy checking enjoying better efficiency still holds true for this special case.

Table 2 shows the number of additional instructions executed for integrity checks of the canaries when the benchmarking programs are running the same workload. Results are consistent with those shown in Figure 5, which, again, confirms our intuition that lazy checking enjoys better efficiency in general.

Table 2. Instructions added for canary integrity check

Program	Reading		Storing		Branch evaluation	
	#	%	#	%	#	%
`bzip2`	4.56E+09	12.17	1.32E+09	3.53	3.49E+09	9.32
`gcc`	8.91E+09	3.93	6.22E+09	2.74	5.39E+09	2.38
`mcf`	1.06E+08	0.28	5.02E+07	0.13	3.57E+07	0.09
`sjeng`	2.54E+11	38.36	2.39E+11	36.04	2.19E+11	33.01
`libquantum`	3.29E+11	5.89	3.29E+11	5.90	3.28E+11	5.89
`astar`	1.63E+10	6.66	9.79E+09	4.01	7.75E+09	3.17
`namd`	1.60E+08	0.00	1.44E+08	0.00	8.50E+07	0.00
`soplex`	3.17E+07	0.01	3.59E+06	0.00	1.65E+07	0.00
`lbm`	2.40E+07	0.00	9.99E+06	0.00	1.29E+06	0.00

The percentage of additional instructions executed for some programs, such as `namd`, `soplex`, and `lbm` is around 0%. However, the runtime overhead for them is still about 2% to 3% as shown in Figure 5. This is because the number of additional instructions executed cannot be ignored although the percentage number is small. Executing these additional branches still produces runtime overhead, but the overhead is small.

4.3 Memory Usage Overhead

Table 3 shows the memory overhead of our experiments with the benchmarking programs in terms of the number of bytes and percentage. As shown, the memory overhead ranges from 24 bytes to 5,220 bytes with an average of 960 bytes, which is about 5.3%. We find such memory usage overhead acceptable.

Note that the memory usage overhead is proportional to the number of sensitive variables statically found in the program and not dynamically related to the specific workload. For example, `lbm` contains only two sensitive variable, which

Table 3. Memory Usage Overhead

Program	Memory Overhead (Bytes)	Percentage (%)
`bzip2`	224	0.01
`gcc`	5220	5.76
`mcf`	52	18.31
`sjeng`	496	0.18
`libquantum`	64	6.45
`astar`	108	1.89
`namd`	1632	0.13
`soplex`	820	5.87
`lbm`	24	9.09
`average`	960	5.3

result in 24 bytes of memory overhead; however, its runtime overhead is still noticeable at 1.3% with some specific workload, as shown in Figure 5.

Security Evaluation on the RIPE Benchmark Having shown that SafeStack$^+$ enjoys reasonably small runtime and memory overhead, we now turn to the security evaluation. First, we want to make sure that SafeStack$^+$ is no worse than SafeStack in defending against control-flow hijacking attacks. For this purpose, we use the RIPE [29] benchmark that contains 850 exploits that attempt to perform control-flow hijacking attacks. Table 4 summarizes the evaluation results under three different settings.

Table 4. Statistical results on RIPE Benchmark

System Name	# of success	# of failure
RIPE with ASLR	130	720
RIPE with ASLR and compiled with SafeStack	80	770
RIPE with ASLR and compiled with SafeStack$^+$	80	770

Our evaluation shows that SafeStack$^+$ and SafeStack enjoys the same advantages in defending against control-flow hijacking attacks (not only the same number of exploits failed but they are the exact same set). Although this result is as expected, it is interesting to observe the consistency of the behavior of these exploits under SafeStack and SafeStack$^+$, i.e., although the unsafe stack has quite different structure, all the exploits behave in the same way on both SafeStack and SafeStack$^+$.

4.4 Security Evaluation on a Data-flow Hijacking Attack

In this section, we use a real-world example to show how SafeStack$^+$ defends against a data-flow attack. This experiment was based on `CVE-2013-0230` on a memory corruption vulnerability for `miniupnpd`.

CVE-2013-0230 reports a buffer overflow bug in miniupnpd before version 1.0. The vulnerability can be exploited by overflowing the stack [4] which results in potentially a control-flow hijacking and a data-flow hijacking scenario. We will show how SafeStack$^+$ defends against the data-flow hijacking attack. Listing 2 presents (part of) the source code of miniupnpd 1.0, with line 11 showing a stack-based buffer overflow if methodlen is more than 2048 bytes long.

Listing 2. ExecuteSoapAction

```
1  ExecuteSoapAction(struct upnphttp * h, const char * action, int n)
2  {
3      char * p;
4      char method[2048];
5      int i, len, methodlen;
6      i = 0;
7      p = strchr(action, '#');
8      methodlen = strchr(p, '"') - p - 1;
9      .......
10     memset(method, 0, 2048);
11     memcpy(method, p, methodlen);
12     syslog(LOG_NOTICE, "SoapMethod: Unknown: %s", method);
13
14     SoapError(h, 401, "Invalid Action");
15 }
```

Figure 6 shows the stack layout when function ExecuteSoapAction is called under SafeStack (left) and SafeStack$^+$ (right), respectively. As we can see, HttpCommand[16] and HttpUrl[128] can be overwritten by method[2048] in SafeStack, which may cause a data-flow hijacking since both HttpCommand[16] and HttpUrl[128] are sensitive variables whose values may affect the execution of conditional branches. However, on SafeStack$^+$, we add canaries for these two variables, which can help detecting the overflow of method[2048] .

We stress that this is a real-world example of vulnerability and the corresponding data-flow hijacking exploits detected by SafeStack$^+$.

5 Limitations and Related Work

In this section, we briefly discuss limitations of SafeStack$^+$ and some related work.

5.1 Limitations

As discussed earlier, the set of sensitive variables we find is an over approximation – a sensitive variable may not be overwritten at all as there are no vulnerable buffers being stored beyond it. We leave it our future work – a more precise static analysis to find the sensitive variables.

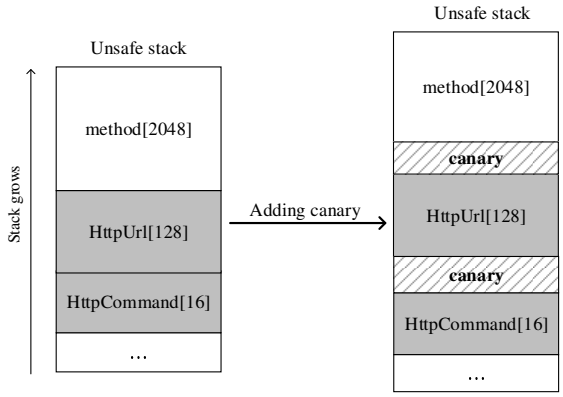

Fig. 6. Layout of Unsafe Stack

A simple idea of attacking SafeStack$^+$ is to brute force the size and value of the canary. With a canary size of 4 bytes, the expected number of tries the attack has to make is 2^{31}. Having a canary of bigger size or multiple canaries (for multiple sensitive variables) makes the attack even more impractical. Memory leakage attacks are possible but very difficult since canaries are only accessible through a dedicated segment register (`%gs`).

Now we have to find the data-flow vulnerability described in this paper manually, which is time consuming. We leave it our future work – automatically find the data-flow vulnerability that can overwrite the data which could potentially affect evaluation of branching conditions.

5.2 Related Work

Many bounds checking methods are proposed to fight against memory corruptions. Cyclone [21] and CCured [25] fuse pointer values and associated bounds meta information into one unified object. With this, bounds information can be read directly from this object and this information can be used for bounds checking instrumentation. SoftBound [24] and Baggy Bounds Checking [7, 15] store the bounds meta information in a shadow space or shadow memory that is separated from the main memory of the program. Shadow memory has better binary compatibility as the layout of objects in main memory is not changed. LowFAT [16] extends the low-fat pointer to stack objects by using pointer mirroring and memory aliasing.

StackGuard [13] patches `gcc` to add a canary before every return address and checks the value of the canary before a function returns. StackGuard ensures targets of return instructions are not overwritten, while SafeStack$^+$ ensures that both targets of return instructions and path sensitive variables are not overwritten. PointGuard [12] encrypts pointers when they are in memory, and decrypts encrypted pointers when they are loaded into CPU registers. PointGuard is similar to SafeStack$^+$ with the main difference being that PointGuard needs to

check when each pointer is loaded into register, which may produce a high runtime overhead. SafeStack$^+$ just checks path sensitive variables when they are loaded from memory (stored into memory or before the execution of conditional branches). Therefore, the performance overhead is much smaller.

Address Space Layout Randomization (ASLR) [3] randomizes the base addresses of the text segment, data segment, stack, and heap at load time. Software diversity [28, 20, 18] implements fine-grained code randomization to mask important details of a program. StackArmor [11] focuses on stack layout randomization. It disrupts the traditional stack organization by making the stack frames and vulnerable buffers neither temporally nor spatially adjacent in memory.

Control Flow Integrity (CFI) [6] ensures that the targets of all indirect branches point to legitimate locations determined statically. However, getting all precise targets for each indirect branch statically is difficult, so many coarse grained CFI methods are proposed [31, 26, 23] to simply include every function in a program in the set of valid targets. CFI could not guarantee protection against all control flow hijacking attacks. Recent results [10, 19, 27] show that many existing CFI solutions can be bypassed in a principled way.

Shadow stack techniques [9, 17, 30, 14] split the stack into two parts: a shadow stack for storing sensitive data such as return addresses and the main stack for storing everything else. SafeStack [22] can be seen as one special case of shadow stack. It stores local variables (called unsafe variables) that may cause memory error onto one unsafe stack (shadow stack), and return addresses and other safe variables are placed onto the main stack. However, this strategy alone does not prevent unsafe variables from attacking each other.

6 Conclusion

This paper presents SafeStack$^+$, which extends SafeStack to make it can defend against both control flow and data flow hijacking attacks. We show that the average runtime and memory overhead of SafeStack$^+$ are 3.0% and 5.3% respectively. In addition, we evaluate how different checking locations would affect the runtime overhead. Results show that, for most programs, checking at memory related operations experiences more runtime overhead and adds more instructions. The security evaluation shows SafeStack$^+$ can effectively counter against both control flow and data flow hijacking attacks.

References

1. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.
2. SafeStack.cpp. http://llvm.org/docs/doxygen/html/SafeStack_8cpp_source.html.
3. Pax ASLR (Address Space Layout Randomiztion). https://pax.grsecurity.net/docs/aslr.txt, 2003.
4. MiniUPnPd 1.0 Stack Buffer Overflow Remote Code Execution. http://www.cvedetails.com/cve/cve-2013-0230, 2013.
5. The LLVM Compiler Infrastructure. http://llvm.org/, 2016.

6. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
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 *Proceedings of the 18th USENIX Security Symposium*, pages 51–66, 2009.
8. S. Andersen and V. Abella. Data execution prevention. *Changes to functionality in microsoft windows xp service pack*, 2, 2004.
9. S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
10. N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, volume 14, 2014.
11. X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security*, 2015.
12. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, volume 12, pages 91–104, 2003.
13. 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 *Proceedings of the 7th USENIX Security Symposium*, volume 98, pages 63–78, 1998.
14. T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.
15. B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell. Baggy bounds with accurate checking. In *International Symposium on Software Reliability Engineering Workshops*, pages 195–200. IEEE, 2012.
16. G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *Symposium on Network and Distributed System Security*, 2017.
17. Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.
18. J. Fu, Y. Lin, and X. Zhang. Code reuse attack mitigation based on function randomization without symbol table. In *Proceeding of the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 394–401. IEEE, 2016.
19. E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.
20. J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 33th IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.
21. T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.
22. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, volume 14, pages 147–163, 2014.

23. Y. Lin, X. Tang, D. Gao, and J. Fu. Control flow integrity enforcement with dynamic code optimization. In *International Conference on Information Security*, pages 366–385. Springer, 2016.
24. S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Conf. on Programming Language Design and Implementation*, 44(6):245–258, 2009.
25. G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, volume 37, pages 128–139. ACM, 2002.
26. M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.
27. F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *International Workshop on Recent Advances in Intrusion Detection*, pages 88–108. Springer, 2014.
28. R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.
29. J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. Ripe: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2011.
30. Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 429–438. IEEE, 2006.
31. M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium*, volume 13, 2013.