Assessing the Impact of Efficiently Protecting Ten Million Stack Objects from Memory Errors Comprehensively

Kaiming Huang
Penn State University
kzh529@psu.edu

Jack Sampson
Penn State University
jms1257@psu.edu

Trent Jaeger
Penn State University
trj1@psu.edu

Abstract*—Despite extensive research on defenses, exploitations on stack memory errors remain a major concern. Previous work has focused primarily on protecting code pointers (e.g., return addresses), but stack data may be compromised due to spatial, type, and temporal memory errors. Recent work on the DATAGUARD system proposes an efficient defense for protecting a significant fraction of stack data from memory errors comprehensively. In this paper, we present an evaluation of DATAGUARD that encompasses several key aspects. Firstly, We assess its applicability and scalability by deploying it on 1,245 packages in Ubuntu 20.04. Secondly, we examine DATAGUARD's effectiveness in identifying and protecting stack data on the evaluation dataset - results show that DATAGUARD is able to protect 12.5 million stack objects, which is around 86% of the total stack objects in these packages. Thirdly, we examined the security enhancements offered by DATAGUARD by evaluating the fraction of protected control data, system calls, and function parameters, as well as the mitigation of real-world CVE exploits. Lastly, we compared the protection of DATAGUARD to CCured and Safe Stack, which shows that DATAGUARD greatly increased the number and fraction of safe stack objects on the analyzed Linux packages. The overall evaluation of DATAGUARD demonstrates the capability of achieving more comprehensive protection with low cost from enforcing lightweight isolation, thus enabling practical adoption to protect software against exploitations on stack memory errors in production environments.

***Index Terms*—Computer security, Software Security, Program Analysis, Memory Errors, Experimental Evaluation.**

I. INTRODUCTION

Memory errors have been arguably the greatest cause of vulnerabilities in software [1]. Programs written in languages that are not memory safe, such as C and C++, may have flaws that allow memory accesses that do not comply with the semantics of the referent (i.e., object to which the pointer was initially assigned), such as the size of its memory region, its lifetime, and/or its data type. If adversaries can control such illicit memory accesses, they can access unauthorized memory regions, stale memory, and/or cause values to be misinterpreted (e.g., use a data value as a pointer) to launch a variety of exploits, such as buffer overflows, use-after-frees, and type confusion. Recent surveys from Google and Microsoft indicate that approximately 70% of flaws in production software occur due to memory safety errors [2], [3].

Researchers have been aware of such flaws since the Anderson report [4] and they have been exploited in-the-wild since the Morris worm [5]. However, despite a tremendous amount of research on defenses, very little program data is currently protected from memory errors. Traditionally, only return addresses stored on the stack are validated before use [6], although recent hardware mechanisms [7] enforce a shadow stack [8] and restrict the values of function pointers [9]. The remainder of program data, including all data variables allocated on the stack, are left prone to exploitation via memory errors, even if that data itself is only referenced by memory-safe accesses.

Researchers have classified memory errors into three classes: (1) spatial errors; (2) type errors; (3) temporal errors. Despite research on defenses for these classes of memory errors, defenses have not been adopted in practice to protect stack data. In general, such defenses employ runtime checks to enforce memory bounds [10], [11] (e.g., to limit accesses to the referent's memory region) or detect accesses outside bounds [12], [13], [14], [15], validate use of expected types in accessing memory regions [16], [17], [18], [15], and prevent accesses to stale memory [19], [20], [21]. However, even after removing unnecessary runtime checks [22], [23], overheads of these techniques have been too high for adoption. Moreover, these defenses do not prevent memory errors comprehensively, and often only focus on one dimension.

How did we get to this situation? In the early 2000s, the CCured system [24], [25], [26] found that approximately 90% of pointers were not used in pointer arithmetic (i.e., must be within memory bounds of the referent, spatially safe) and approximately 99% of pointers were never used in type casts (i.e., must only be interpreted as a single type, type safe). As a result, one insight is that we only need to perform runtime memory checks on a small fraction of pointers to enforce spatial safety or type safety. However, the overheads of the proposed defenses for spatial, type, and temporal safety have been too high for adoption, indicating that these unsafe pointers appear to be used in too many operations to enforce memory safety efficiently over all pointers. As a result, stack data is not protected from memory errors, even for the stack data that can only be accessed by safe pointers.

This appears to be an instance of the aphorism that states, "perfect is the enemy of good" [27]. Recent work has shown that an alternative security goal can enforce the protection of many stack objects from memory errors with low overhead. The DataGuard system [28] shows that *comprehensive memory safety validation* can be used to identify the memory objects that must only be accessed in memory-safe operations and that these objects can be protected by isolating them in a separate memory region from objects that may be accessed unsafely. DataGuard is designed to validate memory safety for stack memory objects, isolating those validated to be safe using the Safe Stack [29]. DataGuard introduces new safety validation strategies to increase the number of objects that cannot be accessed using an unsafe memory operation (i.e., assuming the claimed soundness of alias analyses used), finding over 90% of stack objects are comprehensively memory safe. In addition, DataGuard found that more stack objects could be proven memory safe than for CCured [24], [25], [26] and Safe Stack [29], even though the safety validation techniques used in these systems are incomplete (i.e., some unsafe objects are classified as safe). Finally, because such a high fraction of stack objects could be proven safe, the overhead of the safe stack defense was reduced from over 8% (for SafeStack) to 4.3% because of the reduced toggling between stacks.

In this paper, we evaluate our experience using the open-source DataGuard system [30] to harden entire Linux distributions. Specifically, we apply DataGuard to the 1,623 packages in the Ubuntu 20.04 distribution. The aim is to determine the potential security and performance impact of applying comprehensive memory safety validation to the stack objects of an entire Linux distribution. First, we do not modify the DataGuard system to assess its applicability for automated hardening, finding that 1,245 packages that can be hardened out of the box automatically and assessing the additional features needed to automate the hardening of all packages. Second, we apply DataGuard to compute the fraction of comprehensively safe stack objects across all the packages to which DataGuard applies. We find that 12.5M stack objects out of a total of 14.6M stack objects can be validated as memory safe comprehensively, enabling the protection of 85.4% of stack objects in these packages without runtime checks. Third, we analyze these programs to assess the security impact of protecting these stack objects by collecting their subsequent uses in a variety of security-sensitive operations in these programs. For example, we compute that 413K out of a total of 452K stack objects used in conditional statements are safe from memory errors, which may impact control flows. We also examine the fraction of safe stack objects that are passed to a set of 13 unsafe system calls. Fourth, we assess the performance impact of validating comprehensive memory safety on protecting stack data. Past work found that the performance overhead of the Safe Stack depends on the number of functions in which all local variables are either safe or unsafe. We find that 747K functions out of 1,153K functions total have only safe local variables, which implies that the overhead of the protection should be comparable to the 4.3% measured in the DataGuard paper [28].

II. Stack Memory Safety Background

There are three ways in which memory safety may be violated in unsafe languages such as C and C++.

- **Spatial Safety Errors**: A memory operation may access a location outside the memory region of the referent to illicitly read (e.g., buffer over-read or disclosure) or write (e.g., buffer overflow) an object other than the one assigned as the referent.
- **Type Safety Errors**: A memory operation may access a location using a different data type than that of the referent (or referent's field) assigned to pointer (e.g., enabling a data value to be referenced as a pointer).
- **Temporal Safety Errors**: A memory operation may use a referent outside of the scope of its assignment, such as prior to the referent's assignment (e.g., use-before-initialization) or after deallocation (e.g., use-after-free), to access another object in that memory location.

Stack vulnerabilities of each class have been reported recently (see examples in DataGuard paper [28]), including some even more recent CVEs we identify in Section VI.

Preventing such errors has been a difficult challenge. Researchers found that a large fraction of referents could never violate spatial or type safety because they are not used in pointer arithmetic or type casting operations, respectively [24], [25], [26]. However, there still remained a significant number of memory operations that may result in spatial and type safety errors, and these works did not consider temporal safety errors.

Researchers proposed techniques to prevent or detect memory safety errors using runtime checks, but even optimized runtime defenses are not yet practical for the scope of objects protected. First, a variety of defenses have been proposed for spatial safety errors [10], [11], [12], [15], [31], but even after removing checks for objects that can be proven not to exceed bounds [11], the runtime overhead remains significant. Sanitizers are generally more efficient at detecting spatial memory errors using invalid memory regions (i.e., red zones), but these regions may be bypassed by attackers in some cases. Second, researchers have also proposed techniques to prevent type safety errors [16], [17], [32], [33], [18], [15], which can be applied only when type casts may occur. Since type casts are much less common than pointer arithmetic these defenses have the better performance, but only address one dimension of safety. Temporal safety defenses [19], [20], [21], [34] often perform simple operations, such as zeroing pointers after freeing them to avoid use-after-free exploits, but the number of such operations still introduce significant overheads.

Alternatively, researchers have explored using memory isolation to protect stack data from prevent memory errors. Multistacks have been proposed [35] to isolate stack objects with different security requirements. The safe stack defense [29] isolates stack objects that are referenced from the stack pointer only (i.e., not used in address-taken operations) to protect code pointers and return addresses. However, stack data objects are largely left unprotected. With the emergence of data-oriented attacks [36], [37], [38], there is an urgent need to protect stack data, including data/code pointers, comprehensively.

III. Comprehensive Memory Safety via Isolation

DataGuard [28] was motivated by an observation made over 20 years ago in the CCured system [24], [25], [26] that a preponderance of pointers in C programs never perform any memory operations that could violate memory safety. First, to cause a spatial error a pointer must be used in pointer arithmetic operation, but approximately 90% of pointers are never used in pointer arithmetic. Second, to cause a type error, a pointer must be involved in a type cast operation, but approximately 99% of pointers are never type cast. CCured did not consider temporal safety, but one would expect that assignments that would violate temporal safety for stack pointers (e.g., assigning a stack pointer to the heap or returning a stack pointer when its referent goes out of scope) should be uncommon. Despite many stack pointers being safe from memory errors, we still do not protect much stack data.

Motivated by the observations of the CCured system and stack isolation defenses from multistacks [35], [29], DataGuard asks the following questions. What fraction of stack objects are only aliased by pointers that must only perform memory-safe operations? What are the security and performance impacts of isolating those stack objects from accesses that may violate memory safety on a separate safe stack? Ideally, the goal is to maximize the number of stack objects that can be validated as only having alias that performs memory-safe operations for all classes of memory errors. By isolating these objects from those whose memory accesses may violate memory safety, e.g., using multistacks, the stack objects that pass safety validation can be protected from memory errors comprehensively. Since isolating stack objects does not require runtime checks for memory accesses to the safe objects, the performance overhead of their protection can be reduced.

To maximize the number of stack objects that can be proven as having only aliases whose accesses are safe from memory errors, DataGuard provides static memory safety validation methods for spatial, type, and temporal safety for stack objects. A challenge is that accurate static analysis can be complex and expensive, so DataGuard performs this static analysis in three steps for each memory error class.

First, DataGuard performs simple static analyses to identify the pointers that may perform memory operations that may violate memory safety (e.g., based on CCured [24], [26], [25] and detecting which pointers may escape a single function), and the objects whose aliases never perform such operations. Promisingly, for the programs evaluated [28], 86% of stack objects are only aliased by pointers that cannot violate spatial memory safety and 88% of stack objects are only aliased by pointers that cannot violate temporal memory safety.

Second, DataGuard performs sound[1] static analyses to validate spatial, type, and temporal safety. Specifically, DataGuard employs value-range analysis [39] for spatial safety validation and a liveness analysis [40] for temporal safety. DataGuard only validates type safety for casts between integer types, aiming to ensure that type casts cannot change integer values for value-range analysis.

Third, DataGuard applies symbolic execution to validate cases found unsafe in static analysis. In practice, overapproximation in static analysis yields false positives. However, since stack objects have a limited lifetime, symbolic execution is often capable of validating that objects cannot be accessed unsafely by following the def-use chains.

	# of Packages	*# of SLOC*
Analyzed	1,245 (76.7%)	266,497,755 (77.8%)
Total	1,623	342,451,612

TABLE I: **Statistics of Linux Packages**

The resulting stack objects found memory safe comprehensively (91% in tested programs) are isolated using the Safe Stack defense [29], where the safe stack is made inaccessible from the regular stack via ASLR. Interestingly, since over 75% of functions tested had only safe stack objects, the performance overhead of using the Safe Stack defense is reduced greatly (i.e., from 11% to 4.3%) because there is less togging between the safe and regular stacks.

IV. Our Investigation Goals

In this paper, we ask several questions about the potential security and performance impact of the DataGuard approach.

First, a question is *whether the DataGuard approach is broadly applicable to Linux systems*. If the technique requires manual effort or only applies for a small number of packages, then its utility is limited. In this case, our goal is to evaluate the entire set of packages of a Linux distribution, Ubuntu 20.04.

Second, a question is *what fraction of stack objects can be protected from memory errors automatically using the off-the-shelf DataGuard system*. The original paper found 91% of stack objects could be protected in the tested program. We wonder what fraction of stack objects can be protected for packages in a Linux distribution and the total number of stack objects protected in these programs.

Third, we assess *the security impact by computing the uses of the protected stack data* to assess whether what fraction of stack data may be used in security-sensitive operations impacting control-flows (e.g., used in conditional tests) and external resources (e.g., used as system call arguments).

Finally, we evaluate *the potential overhead of the applying the DataGuard approach for each program*. While it is impractical to run all programs, we assess performance overhead in terms of the fraction of functions with only safe local variables and passed parameters. If all local variables and passed parameters are safe, then only one (safe) stack is needed for the function. As a result, the stack pointer never needs to be changed (i.e., toggled) between the safe and regular stacks for this function, which is the main cause of performance overhead of the Safe Stack defense.

V. Experimental Setup

Our experiments on DataGuard are conducted on Ubuntu 20.04 with Linux kernel version 5.8.0-44-generic on x86_64 architecture using LLVM 10.0, running on an Intel CPU i9-9900K with 128 GB RAM. We used the original implementation of DataGuard that is open-sourced on Github [30],

[1] DataGuard uses the definition of *soundness* common in the static analysis community, where a sound analysis overapproximates the program's possible executions.

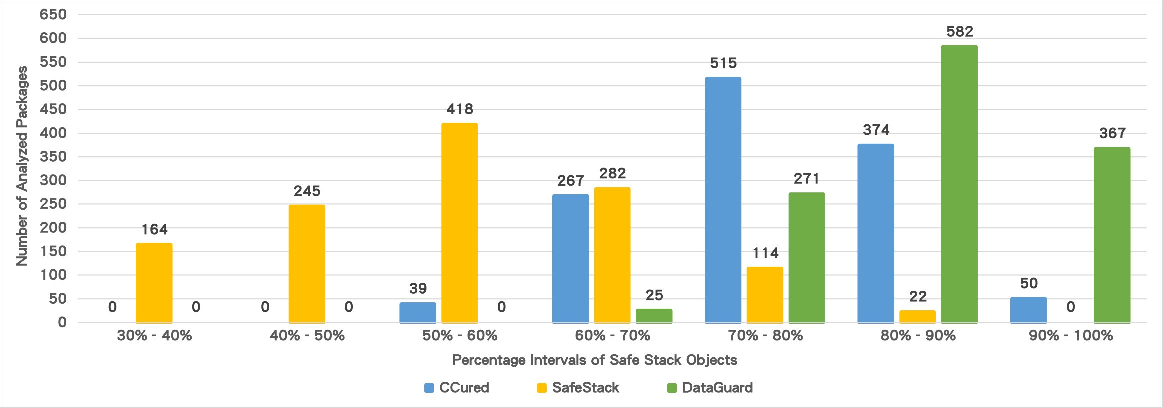

Fig. 1: **Distribution of Packages w.r.t. Fraction of Safe Stack Objects**. The X-axis represents the interval of the fraction of safe stack objects that are protected by corresponding approaches, 0%-30% is omitted since none of the 3 approaches protects less than 30% of any package. The Y-axis represents the number of packages that falls into the corresponding fraction interval of safe stack objects.

	Total	***DataGuard-Safe***
Object	14,627,355	12,484,971 (85.4%)
Control Data	451,839	412,725 (91.3%)
Function	1,152,744	747,391 (64.8%)
Parameter	1,904,262	1,622,867 (85.2%)

TABLE II: **Statistics of DataGuard Analysis on Linux Packages.**

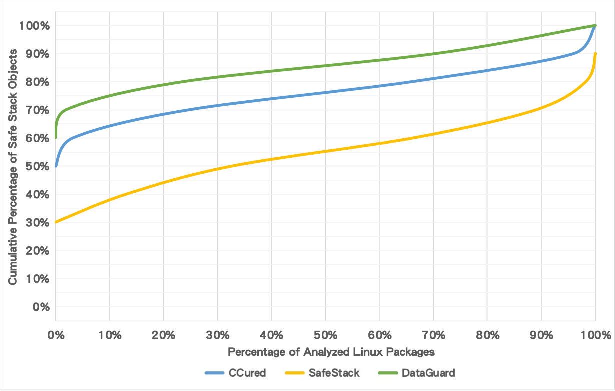

Fig. 2: **Cumulative Distribution of the Fraction of Protected Safe Stack Objects for All Analyzed Linux Packages**. The X-axis represents the percentage of analyzed Linux packages (1,245 in total). The Y-axis represents the percentage of safe stack objects found by DataGuard [28], CCured implementation of Nescheck [41], and SafeStack [42]. The figure can be understood as "(1 - X-axis)% of analyzed packages have ***at least*** Y-axis% of safe stack objects," as indicated by the corresponding three approaches.

including its prerequisite analyses such as PDG [43] and SVF [44]. The list of pre-installed and commonly used packages of Ubuntu 20.04 is obtained from the official Ubuntu Package list [45]. Our experiments utilized the uClibc library as the C/C++ library since it is friendly to and originally used by the DataGuard.

To compile the analyzed Linux packages into LLVM bitcode for deploying DataGuard analyses and protection, we used wllvm [46]. Wllvm is a tool that is commonly used for building whole-program (or whole-library) LLVM bitcode files from an unmodified C or C++ source package. For the Linux Packages to be compatible with wllvm, they must use a configuration file for the compilation process and accept clang-10 as its compiler[2]. Such packages must also be either written solely in C/C++ or support compiling C/C++ code independently if they need to be cross-compiled from multiple languages. Also, The packages must be compatible with the prerequisite analyses toolchain used by DataGuard (e.g., PDG and SVF) for the deployment and evaluation.

VI. Experiment Results

A. Protecting Linux Packages

Table I and II shows the overview summary of our experimental results of DataGuard on Linux Packages. We evaluate DataGuard for the four investigation goals of Section IV.

First, we find that DataGuard successfully analyzed 1,245 packages (76.7%) of the 1,623 packages in total for the Linux Ubuntu 20.04 distribution. This constitutes approximaately 266 million LOC (77.8%) of the 342 million LOC total (see Table I). The remaining 378 packages are not analyzed due to the incompatibility with the requirements mentioned in Section V, including failure in LLVM bitcode generation, and not supported by prerequisite analyses of DataGuard.

Second, among all the analyzed packages, DataGuard protects 12,484,971 out of 14,627,355 million (85.4%) stack objects by validating that these objects have only memory accesses that satisfy spatial, type, and temporal memory safety. The objects are protected from attacks on memory accesses to other objects using the Safe Stack defense [29] by DataGuard. We note that this is a reduction in the fraction of stack objects validated as satisfying memory safety comprehensively in the DataGuard paper (91%), but still a significant fraction of stack data can be protected by construction.

Third, we evaluate the potential security impact of DataGuard from three perspectives: (1) control data, (2) function parameters, and (3) arguments of sensitive system/library calls. The data checked in any conditional branch (i.e., control data)

[2] Note that other approaches to generate bitcode can also be used such as LLVM Gold linker, we use wllvm for simplicity and convenience automation.

can be exploited by attackers to alter control flows of the program, so we evaluate the fraction of control data protected by the DataGuard across all analyzed Linux packages. Of the 451,839 stack objects used as control data total, 412,725 stack objects are classified as safe and isolated from memory errors on other unsafe stack objects, which is approximately 91.3% of these objects. Thus, the attack surface created by stack objects as control data is reduced by DataGuard by an even greater fraction than in general.

We also quantified the security impact of DataGuard on invoking functions in general and specifically for security-sensitive system/library calls. DataGuard protected 1,622,867 of 1,904,262 of total function parameters on the stack for all packages, which is a similar percentage (85.2%) to the fraction of protected stack objects in general. We analyzed 13 system calls that commonly serve as the key/last step of the attacker to hijack control flow or escalate privilege, as listed in Table III. Approximately 90% of calls to these functions take only safe stack objects as arguments. This means there are only a small fraction of cases where attackers can corrupt data on the unsafe stack to invoke sensitive system/library calls with these corrupted arguments. Instead, nearly all the values passed as arguments to these functions are stored on the safe stack, isolated from tampering through memory errors.

Finally, as an indicator of performance, we counted the number of functions that DataGuard validated as having only safe stack objects (i.e., local variables and passed parameters), which removes the overhead caused by switching between stacks, as described above. We found that out of a total 1,152,744 functions, 747,391 of them (64.84%) only contain safe stack objects. While lower than the 76% of safe functions found in the DataGuard application set, this is still significantly more safe functions than CCured (41%), which incurred an 8.6% overhead. Thus, we expect a performance overhead on overage only slightly higher than the 4.3% measured.

B. Distribution of Linux Packages on Safe Stack Objects

Figure 1 shows the distribution of the Linux Packages w.r.t., the fraction of safe stack objects in each package. We compared DataGuard with CCured and SafeStack mechanisms of which DataGuard is built on top and originally compared with [28]. The CCured implementation is adopted from Nescheck approach [41], The SafeStack defense is implemented by Clang [47]. As we can see, the DataGuard system outperforms the other two in identifying and protecting safe stack objects through isolation. A majority of the Linux packages have more than 80% of stack objects protected comprehensively from memory safety errors under the protection of DataGuard, whereas CCured and SafeStack have peak protection fractions in the intervals 70%-80% and 50%-60%, respectively.

The cumulative distribution (Figure 2) also shows a similar finding. The percentage of safe stack objects protected by DataGuard is obviously more than CCured and SafeStack approaches. Specifically, DataGuard protects more than 70% of the safe stack objects for 97% of the packages (i.e., 1-3% in the figure) and more than 80% of the safe stack objects for 75% (i.e., 1-25%) of the packages.

	Safe	***Total***
exec	10,362 (91.82%)	11,285
execve	425 (89.85%)	473
system	2,933 (83.46%)	3,514
popen	587 (92.73%)	663
fopen	11,765 (94.00%)	12,516
dlopen	1,182 (86.84%)	1,361
link	17,764 (81.18%)	21,881
symlink	1,341 (92.85%)	1,445
read	134,095 (85.62%)	156,608
write	89,271 (94.01%)	94,957
strcpy	17,694 (82.23%)	21,519
strncpy	8,561 (92.56%)	9,249
memcpy	87,462 (85.39%)	102,424
AVERAGE	***383,442 (87.56%)***	***437,895***

TABLE III: **Safety of Sensitive System or Library Calls**

C. Mitigating Real-world Exploits

Figure 3 shows a CVE caused by a stack-based buffer overflow (i.e., spatial error). The vulnerability CVE-2023-2837 [48] was discovered in May 2023 in a modular multimedia framework `gpac` prior to 2.2.2. Gpac is commonly used to process, inspect, package, stream, playback, and interact with media content, and it is officially supported by many primary streaming services. Gpac can be installed on Ubuntu as prebuilt binaries through package manager `apt-get`.

Function `xml_sax_parse` utilizes an index variable `current_pos` at line 11 to traverse the data saved in `parser->buffer` on the stack and copy the data into the heap memory pointed by stack pointer `orig_buf`. The `parser->buffer` has constant size, but the programmer falsely utilizes another variable (an element of `parser`) `line_size` to perform the size check. Given `current_pos` is increased by 1 along with the execution of the loop each time. This can result in stack-based buffer overflow of `parser->buffer` if `line_size` is corrupted.

DataGuard successfully mitigates the CVE by classifying `parser->buffer` as unsafe, isolating its accesses from objects on the safe stack. The reason for such a classification is that, though `parser->buffer` is declared with constant size, the access range of it depends on a variable `parser->line_size` whose value cannot be concretized statically, (i.e., the value range analysis of DataGuard cannot validate the `current_pos` is incremented to a constant limit). Thus, since the access range of `parser->buffer` is not bounded by a constant within the size of `parser->buffer`, DataGuard classifies `parser->buffer` as unsafe. The isolation of `parser->buffer` prevents the overflow from corrupting other stack objects.

Figure 4 shows a recent CVE of stack-based integer overflow enabled by unsafe type cast (i.e., type error). The vulnerability CVE-2023-2610 [49] was discovered in May 2023 in `vim` prior to 9.0.1532. Vim is a highly configurable and powerful text editor designed for efficient text editing. It stands for ”Vi Improved” and is an enhanced version of the original Vi editor. Vim is widely used among programmers, system

```c
1  static GF_Err xml_sax_parse(GF_SAXParser *parser,
2                              Bool force_parse){
3      u32 i = 0;
4      ...
5      while (parser->current_pos < parser->line_size) {
6          ...
7          char *orig_buf;
8          GF_Err e;
9          ...
10         //Unsafe pointer arithmetic
11         orig_buf = gf_strdup(parser->buffer +
12                              parser->current_pos);
13         ...
14
15         e = gf_xml_sax_parse_intern(parser,
16                                     orig_buf);
17         gf_free(orig_buf);
18         parser->current_pos+=1+i; //no size check
19     }
20 }
```

Fig. 3: **CVE-2023-2837**. Attacker can overflow parser->buffer at line 11 due to wrong size check and offset increment at line 5 and line 18.

administrators, and other professionals who work with text files. Vim can be installed on Ubuntu as prebuilt binaries through package manager apt-get.

The function regtilde declared two integer variables len and prevlen to track the offset of source and destination buffer of memory writes by mch_memove at line 19 and 20. However, two unsafe type casts occur at line 14 and 18. At line 14, the signedness of return value of STRLEN is changed due to the cast from type size_t to int. Normally, casting from an unsigned integer (i.e., size_t) to a signed integer (i.e., int) will not cause problems. Unfortunately, in this case, the variable prevlen is then used to calculate the offset of the destination buffer tmpsub in the memory write at line 25 by calling STRCPY. If prevlen is assigned to a negative number resulting from the type cast by feeding a really long reg_prev_sub, it can cause access (write) to unauthorized memory and/or lead to unexpected crashes of the program, since the offset is calculated by tmpsub+len.

Similarly, in line 18, the result of the pointer subtraction of type char_u is cast into int. Though it is less likely to cause a problem than the previous case, there is still a window for attacker to corrupt the type cast that leaves len a negative value. The variable len is then used to calculate the offset of the destination buffer tmpsub in the memory write at line 20 by calling mch_memove, and line 25 by calling STRCPY, resulting in the same corruption as above.

DataGuard's value-range analysis successfully identifies that the prevlen and len suffer from integer type casts that change their signedness. Also, the value-range analysis finds that the destination buffer tmpsub of the memory writes at the line 20 and 25 involves in access range that based on the value of a variable (i.e., access range is not constant and cannot be concretized statically). Thus, the affected buffer tmpsub is classified as unsafe and placed on the unsafe stack. Its corruption due to this CVE will not affect other safe objects isolated on the safe stack.

```c
1  static char_u  *reg_prev_sub = NULL;
2
3  char_u *regtilde(char_u *source, int magic){
4      char_u  *newsub = source;
5      char_u  *tmpsub;
6      char_u  *p;
7      int     len;
8      int     prevlen;
9
10     for (p = newsub; *p; ++p){
11         ...
12         if (reg_prev_sub != NULL){
13             // length = len(newsub)-1+len(prev_sub)+1
14             prevlen = (int)STRLEN(reg_prev_sub);
15             tmpsub = alloc(STRLEN(newsub) + prevlen);
16             if (tmpsub != NULL){
17                 // copy prefix
18                 len = (int)(p - newsub);
19                 mch_memmove(tmpsub, newsub, (size_t)len);
20                 mch_memmove(tmpsub + len, reg_prev_sub,
21                                 (size_t)prevlen);
22                 // copy postfix
23                 if (!magic)
24                     ++p; // back off backslash
25                 STRCPY(tmpsub + len + prevlen, p + 1);
26                 ...
27                 newsub = tmpsub;
28                 p = newsub + len + prevlen;
29             }
30         }
31         ...
32     }
33     ...
34     reg_prev_sub = vim_strsave(newsub);
35     return newsub;
36 }
```

Fig. 4: **CVE-2023-2610**. Attacker is able to access unauthorized memory through memory writes at line 20 and 25, which is cased by integer type cast that changed signedness at line 14 and 18.

VII. Conclusions

There is an urgent need for comprehensive and lightweight protection on stack memory that can be suitably deployed in production environments. To advance this goal, this paper provided an evaluation on DataGuard, which is a recently proposed and open-source framework for identifying and isolating safe stack objects from illicit accesses through exploitations of stack memory errors. The evaluation process provided an in-depth examination of DataGuard in several key aspects, including practical implementations, effectiveness, scalability, security enhancement, and performance overheads by deploying and testing it on 1,245 Linux packages for Ubuntu 20.04. By critically assessing these factors, we found that DataGuard is able to effectively protect around 85.6% of the stack objects from being tampered with, thereby substantially mitigating the potential exploitation of memory errors of all analyzed Linux packages. Furthermore, we examined the possible security impact of DataGuard on protecting critical and sensitive data, and the ability to mitigate real-world exploits. Compared with previous works such as CCured and Safe Stack, DataGuard is a more comprehensive and efficient protection framework against stack memory errors that offers protection of an increased number of stack objects and scope that covers all memory error classes, with the lightweight isolation and the removal of runtime checks for safe stack objects. This evaluation serves as a valuable foundation for future research, enabling progress and innovation in the pursuit of optimal solutions for protecting stack objects from memory errors comprehensively and efficiently.

Acknowledgements

We would like to thank our anonymous reviewers for the invaluable guidance on revision of this paper and their insightful feedback. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2- 0045 (ARL Cyber Security CRA) and National Science Foundation grant CNS-1801534. Any views, opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

References

[1] NSA-CSS, "Nsa releases guidance on how to protect against software memory safety issues," https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/, 2022.

[2] A. Taylor, A. Whalley, D. Jansens, and N. Oskov, "An update on memory safety in chrome," https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, 2021, accessed on May 28, 2023.

[3] Microsoft, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019, accessed on May 28, 2023.

[4] J. P. Anderson, "Computer Security Technology Planning Study, Volume II," Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), Tech. Rep. ESD-TR-73-51, October 1972.

[5] D. Seeley, "A Tour of the Worm," https://www.cs.unc.edu/∼jeffay/courses/nidsS05/attacks/seely-RTMworm-89.html.

[6] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USA: USENIX Association, 1998, p. 5.

[7] "Control-flow Enforcement Technology," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[8] wikipedia, "Shadow Stack," https://en.wikipedia.org/wiki/Shadow_stack, Last Edited 03-02-2023.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 245–258. [Online]. Available: https://doi.org/10.1145/1542476.1542504

[11] 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 Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 51–66.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[13] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," *CoRR*, vol. abs/1806.04355, 2018.

[14] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *CGO '15*, 2015.

[15] G. J. Duck and R. H. C. Yap, "Effectivesan: Type and memory error detection using dynamically typed c/c++," *SIGPLAN Not.*, vol. 53, no. 4, p. 181–195, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192388

[16] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 81–96.

[17] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for c++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2373–2387. [Online]. Available: https://doi.org/10.1145/3133956.3134062

[18] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 517–528. [Online]. Available: https://doi.org/10.1145/2976749.2978405

[19] E. V. D. Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable Use-after-free Detection," *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[20] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing Use-after-free with Dangling Pointers Nullification," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: The Internet Society, 2015.

[21] Y. Younan, "FreeSentry: Protecting Against Use-after-free Vulnerabilities Due to Dangling Pointers," in *22nd Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, 2015.

[22] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "{SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 479–494.

[23] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *Usenix Security Symposium*, 2022.

[24] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, p. 477–526, may 2005. [Online]. Available: https://doi.org/10.1145/1065887.1065892

[25] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," *SIGPLAN Not.*, vol. 37, no. 1, p. 128–139, jan 2002. [Online]. Available: https://doi.org/10.1145/565816.503286

[26] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," *SIGPLAN Not.*, vol. 38, no. 5, p. 232–244, may 2003. [Online]. Available: https://doi.org/10.1145/780822.781157

[27] Voltaire, "Questions sur l'encyclopédie, par des amateur," https://archive.org/details/questionssurlenc02volt/page/250/mode/2up, 1770.

[28] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "The taming of the stack: Isolating stack data from memory errors," in *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: The Internet Society, 2022.

[29] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 147–163.

[30] K. Huang, "Dataguard opensource on github," https://github.com/Lightninghkm/DataGuard, 2022, accessed on 2023.

[31] Duck, Yap, and Cavallaro, "Stack Bounds Protection with Low Fat Pointers," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[32] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "Vtrust: Regaining trust on virtual calls," in *Symposium on Network and Distributed System Security (NDSS)*. San Diego, CA, USA: The Internet Society, 2016.

[33] LLVM, "Clang undefined behavior sanitizer," http://clang.llvm.org/docs/UsersManual.html, 2023, accessed: 2023-05-02.

[34] A. Milburn, H. Bos, and C. Giuffrida, "Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *Network and Distributed System Security Symposium*, 2017.

[35] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended Protection against Stack Smashing Attacks without Performance Loss," in *2006 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[37] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[38] L. Cheng, S. Ahmed, H. Liljestrand, T. Nyman, H. Cai, T. Jaeger, N. Asokan, and D. D. Yao, "Exploitation techniques for data-oriented attacks with existing and potential defense approaches," *ACM Trans. Priv. Secur.*, vol. 24, no. 4, sep 2021. [Online]. Available: https://doi.org/10.1145/3462699

[39] A. Simon, "Value-range analysis of c programs: Towards proving the absence of buffer overflow vulnerabilities," 2008.

[40] Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. Reeves, "Fast Copy Coalescing and Live-range Identification," in *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation. (PLDI)*, 2002.

[41] D. Midi, M. Payer, and E. Bertino, "Memory safety for embedded devices with nescheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 127–139. [Online]. Available: https://doi.org/10.1145/3052973.3053014

[42] "Safe Stack - Clang 12 documentation," https://clang.llvm.org/docs/SafeStack.html.

[43] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2359–2371. [Online]. Available: https://doi.org/10.1145/3133956.3134066

[44] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[45] Ubuntu, "Ubuntu packages search," https://packages.ubuntu.com/, accessed on 2023.

[46] I. A. Mason, "Whole program llvm," https://github.com/travitch/whole-program-llvm, accessed on 2023.

[47] "Clang Documentation - SafeStack," Clang document at https://clang.llvm.org/docs/SafeStack.html, 2020.

[48] "Cve-2023-2837," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2837, accessed on 2023.

[49] "Cve-2023-2610," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2610, accessed on 2023.