Grenoble INP | **UGA**
Master CybserSecurity | **Year 2022-23**

Software Security & Secure Programming

Written Exam - Tuesday January the 17th, 2023

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Electronic devices are forbidden.

This exam contains two distinct parts:

1. One exercise, supposed to be solved in less than 1h;
2. Some questions on a research paper, allow about 1h to read the paper and answer these questions.

Exercise. (∼ 10 pts)

We consider the following C program (on the left) and its Control Flow Graph (CFG, on the right):

```c
#define N 5

int main () {
  int x, y;
  int T[N] ;

  x = 0 ;
  y = 0 ;
  while (x<N+1) {
     if (x%2==1)
              T[x] = y ;
     x = x + 1 ;
     y = y + 100 ;
  } ;
  return 0 ;
}
```

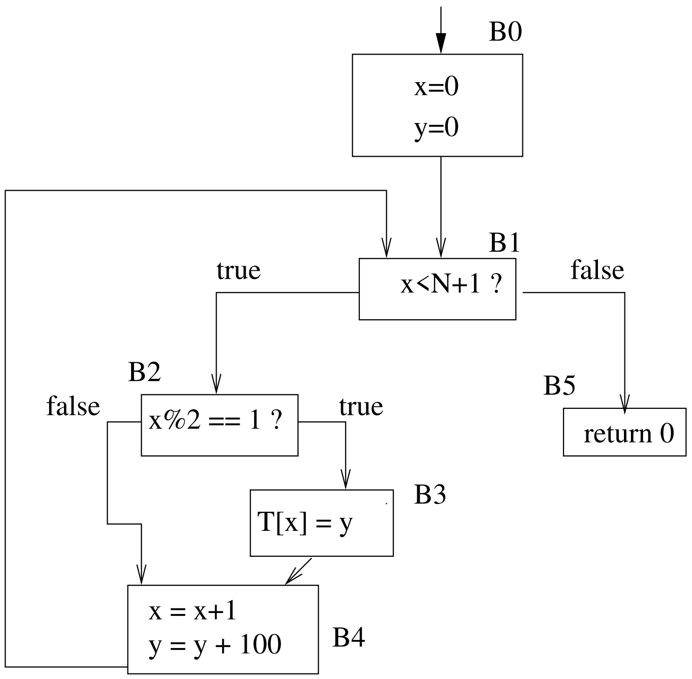
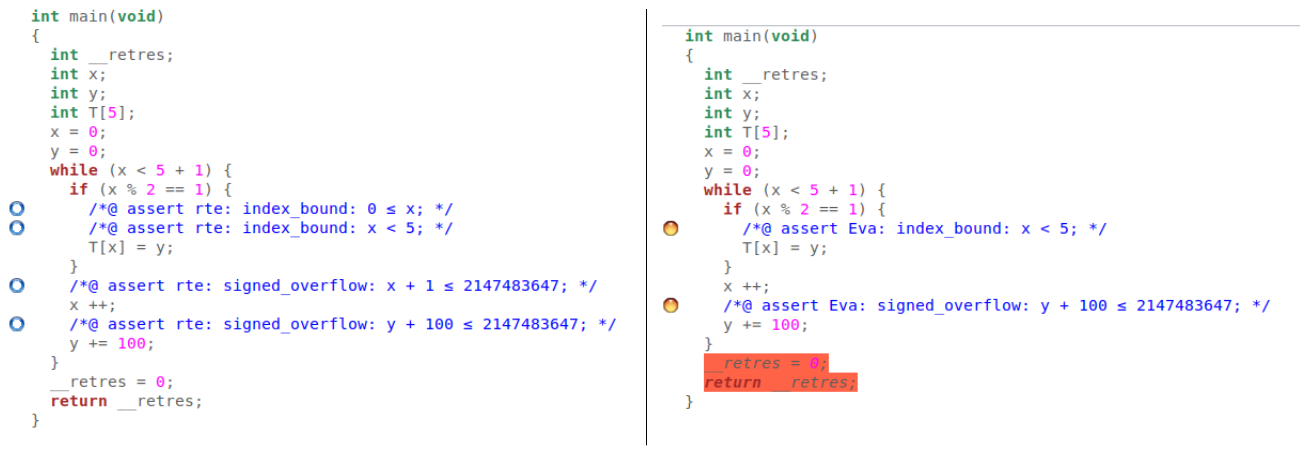


Figure 1: a C program and its Control-Flow Graph (CFG)

Q1. Figure 2 (left) is a screen copy of the results produced when running the `-rte` (runtime-error) option of Frama-C. Explain these results.

Q2. Figure 2 (right) is a screen copy of the results produced when running the `-eva` (value-set analysis) option of Frama-C. According to Frama-C what is (potentially) wrong with this program ?



```
int main(void)
{
  int __retres;
  int x;
  int y;
  int T[5];
  x = 0;
  y = 0;
  while (x < 5 + 1) {
    if (x % 2 == 1) {
      /*@ assert rte: index_bound: 0 ≤ x; */
      /*@ assert rte: index_bound: x < 5; */
      T[x] = y;
    }
    /*@ assert rte: signed_overflow: x + 1 ≤ 2147483647; */
    x ++;
    /*@ assert rte: signed_overflow: y + 100 ≤ 2147483647; */
    y += 100;
  }
  __retres = 0;
  return __retres;
}
```

```
int main(void)
{
  int __retres;
  int x;
  int y;
  int T[5];
  x = 0;
  y = 0;
  while (x < 5 + 1) {
    if (x % 2 == 1) {
      /*@ assert Eva: index_bound: x < 5; */
      T[x] = y;
    }
    x ++;
    /*@ assert Eva: signed_overflow: y + 100 ≤ 2147483647; */
    y += 100;
  }
  __retres = 0;
  return __retres;
}
```

Figure 2: Results obtained with Frama-C when running `-rte` (left) and `-eva` (right)

Q3. We would like to know if the results produced by the value-set analysis of Frama-C are *false positives*. Compute (manually) a value-set analysis using intervals, and give the abstract values obtained for variables `x` and `y` at the entry and exit locations of each basic block (as Frama-C did when running its value-set analysis). You can use widening/narrowing operators if you want (but it is not mandatory on this example). According to this computation, is your conclusion similar to the one produced by Frama-C ?
According to your own understanding of this program behavior at runtime, is there any false positive here ?

Q4. We now define the constant `N` with the value 1000. What would be the abstract values obtained when doing a value-set analysis (with widening and narrowing) for variables `x` and `y` at each entry and exit locations of each basic blocks ? Which run-time errors are detected by this analysis ? Which ones are false positives ?

Q5. We now want to know, using *symbolic execution*[1], **under which conditions for the constant** `N` the code of Figure 1 does contain a **buffer overflow**. To answer you should:

- explain how you would proceed in practice, including how you would modify the original code (if necessary);
- give an example of a *r*elevant path predicate you expect to obtain;
- tell if this path predicate is satisfiable;
- and finally conclude about the "soundness" of the results you will obtain at the end (i.e., how much they are "trustful").

[1]e.g., with a tool like PathCrawler

Questions on a research paper. (∼ 10 pts)

Read **sections I to V** of the paper given in appendix, answering the following questions as long as you read it. The objective of this part of the exam is to evaluate your ability to understand a description of a proposed vulnerability mitigation technique (its strengths and limitations, with respect to other approaches you know). When answering the following questions you should not copy entire sentences from the paper but rather illustrate your point with examples and comments from your own.

Sections marked with a **vertical line** in the margin can be **skipped**. Sentences which are **underlined** have to be **explained** (see questions below).

Q1 [section I].

1. According to you, why do we still find CVEs related to stack memory errors ?
2. What are the main weaknesses of the stack protections mentioned in this section ?
3. *“. . . the emergence of return-oriented attacks [69] raised concerns about protecting the integrity of code pointers on the stack in general, motivating new defenses . . . ”*

   Can you better explain this sentence ?
4. What are the main objectives of the work described in this paper ?

Q2 [section II-A].

1. Draw a possible layout for the execution stack corresponding to the code of Figure 1.
2. Explain the 3 error classes, giving concrete examples from your own and showing precisely the attacker gains for each of them.
3. *“ . . . the need to circumvent stack defenses (see Section II-D) has motivated other attack vectors, such as the modification of control data (e.g., line 6 of Figure 1) and exfiltration of sensitive stack data (e.g., line 4 in Figure 1)”*

   Can you better explain this sentence ?

Q3 [section II-C].

1. What is a *safe object* ? And why is it called “safe” ?
2. Explain precisely the purpose and use of the 2 stacks (e.g., by explaining their use on an example of your own).
3. Why *type errors* do not impact return addresses, nor other code pointer variables ?
4. What are the false positives and false negatives introduced by the “Safe Stack” solution ?

Q4 [section III].

1. Explain why `imax`, `omax`, `ibuff` and `obuff` can be considered as *safe objects* and stored in an isolated stack.
2. What are the 2 main code analysis techniques used in DataGuard ?
3. Why is it so important for theses analysis to rely on over-approximations ?

Q5 [section IV].

1. Why ALSR is required for DataGuard ?
2. Why constants are expected to be stored all the time in read-only memory ?

Q6 [section V-A].

1. give a C code example from your own containing a *safe* and a *non-safe* object (according to Definition 1) ?
2. Why pointer arithmetic is dangerous for spatial memory errors ?
3. *"If any pointer is aliased by a pointer passed from a caller or that references a heap or global object, then DataGuard identifies the need for temporal analysis"*

   Can you better explain this sentence ?

Q7 [section V-B]. Back to your example of non safe code object (Question 6.1), explain which constraints are not satisfied (and hence make these objects non-safe).

Q8 [section V-C]. Mention a tool you are aware of which relies on *value-range analysis.*

Q9 [section V-D]. Does symbolic execution **over-approximate** the behavior of a program ? So what is "wrong" in this paper ?

Q10 (more general question).

1. What about applying the ideas proposed in this paper to avoid **heap** memory corruptions ?
2. What are the main limitations you can see regarding the proposed approach ?

The Taming of the Stack: Isolating Stack Data from Memory Errors

Kaiming Huang[†], Yongzhe Huang[†], Mathias Payer[‡], Zhiyun Qian[§], Jack Sampson[†], Gang Tan[†], Trent Jaeger[†]
[†] The Pennsylvania State University [‡] École Polytechnique Fédérale de Lausanne (EPFL) [§] University of California, Riverside
[†] {kzh529, yzh89, jms1257, gxt29, trj1}@psu.edu, [‡] mathias.payer@nebelwelt.net, [§] zhiyunq@cs.ucr.edu

Abstract*—Despite vast research on defenses to protect stack objects from the exploitation of memory errors, much stack data remains at risk. Historically, stack defenses focus on the protection of code pointers, such as return addresses, but emerging techniques to exploit memory errors motivate the need for practical solutions to protect stack data objects as well. However, recent approaches provide an incomplete view of security by not accounting for memory errors comprehensively and by limiting the set of objects that can be protected unnecessarily. In this paper, we present the DATAGUARD system that identifies which stack objects are safe statically from spatial, type, and temporal memory errors to protect those objects efficiently. DATAGUARD improves security through a more comprehensive and accurate safety analysis that proves a larger number of stack objects are safe from memory errors, while ensuring that no unsafe stack objects are mistakenly classified as safe. DATAGUARD's analysis of server programs and the SPEC CPU2006 benchmark suite shows that DATAGUARD improves security by: (1) ensuring that no memory safety violations are possible for any stack objects classified as safe, removing 6.3% of the stack objects previously classified safe by the Safe Stack method, and (2) blocking exploit of all 118 stack vulnerabilities in the CGC Binaries. DATAGUARD extends the scope of stack protection by validating as safe over 70% of the stack objects classified as unsafe by the Safe Stack method, leading to an average of 91.45% of all stack objects that can only be referenced safely. By identifying more functions with only safe stack objects, DATAGUARD reduces the overhead of using Clang's Safe Stack defense for protection of the SPEC CPU2006 benchmarks from 11.3% to 4.3%. Thus, DATAGUARD shows that a *comprehensive* and *accurate* analysis can both increase the scope of stack data protection and reduce overheads.

I. INTRODUCTION

Researchers have long wanted to protect stack data from exploitation from memory errors. At least by the time of the "Anderson report" [6] in 1972, researchers acknowledged the possibility of *stack overflow attacks* [64], which exploit process execution by writing beyond the bounds of a stack memory buffer to modify other stack data, particularly return addresses. By modifying a return address, an adversary can control which code will be executed when a function returns. Such attacks have been used in the wild since at least the Morris worm [71] in 1988, including famous worm malware, such as Code Red [30] and SQL Slammer [58].

Despite increased awareness and testing, stack memory errors remain a major threat to software security because stack overflow vulnerabilities remain common and new exploit methods have been discovered. First, improvements in testing for memory errors have not eliminated stack overflow vulnerabilities. Recent stack overflow vulnerabilities (e.g, CVE-2021-28972, CVE-2021-24276, CVE-2021-25178) continue to threaten critical software, such as the Linux kernel. Second, adversaries have found other ways that they can effectively exploit memory errors. For example, adversaries may exploit *out-of-bounds read errors*[1] (e.g., CVE-2021-3444, CVE-2020-25624, CVE-2020-16221) to disclose sensitive stack information (e.g., to circumvent stack defenses), *type errors* (e.g., CVE-2021-26825, CVE-2020-15202, CVE-2020-14147) to reference memory using different type semantics, and *temporal errors* (e.g., CVE-2020-25578, CVE-2020-20739, CVE-2020-13899) to reference memory using stale or uninitialized pointers. Nowadays, many attacks target stack data pointers and data values to exploit programs by circumventing stack defenses to redirect data flows [38] or disclose sensitive data [75], and such attacks can even be generated automatically [40].

Current defenses to prevent the exploitation of stack memory errors have a limited scope and/or remain too expensive for broad deployment. Originally, the focus of stack defenses was only on protecting the integrity of return addresses, such as by using *Stack Canaries* [20] or *Shadow Stacks* [15]. While both defenses can now be enforced with low overhead [4], [11], [92], the emergence of *return-oriented attacks* [69] raised concerns about protecting the integrity of code pointers on the stack in general, motivating new defenses. The *Safe Stack* [45] defense employs a separate stack to store all stack objects whose accesses cannot cause buffer overflows, which protects code pointers from being overwritten. However, as we show, the invariants the Safe Stack defense checks do not prevent other attacks on memory errors, such as type confusion [82] or use-before-initialization [83], that are necessary to prevent exploits on stack objects in general. In addition, said invariants are also too conservative, which causes many stack objects to be handled as if they are unsafe, which leaves objects unprotected unnecessarily and increases the overhead of the Safe Stack defense. Thus, imprecision in identifying safe stack objects creates a lose-lose situation, where security and performance both suffer needlessly.

Alternatively, researchers have explored the design of techniques to prevent exploitation of individual classes of memory errors systematically, but these techniques have been seen to be too expensive to be deployed in practice. For example, techniques to prevent spatial errors validate object bounds on each reference [60], [73], incurring significant overhead even when applied only to stack objects [28]. Researchers

[1]In this paper, we group stack over/underflows and out-of-bounds reads under the term *spatial errors*.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/NDSS.2022.23060
www.ndss-symposium.org

have employed static analysis techniques to remove checks for objects that can be proven to only be accessed safely [3], [61]. While such analyses validate safety requirements that are less conservative than those used for the Safe Stack defense, these static analyses still over-approximate the number of unsafe objects significantly, causing some unnecessary runtime checks to be retained on objects that are actually safe. Other defenses are necessary to prevent exploitation of type [35], [41], [47] and temporal errors [44], [46], [88] that incur additional overhead. In addition, the idea of employing isolation or encryption to provide selective data/memory protection has been proposed recently [65], [67]. However, these techniques do not ensure memory safety for all objects in isolated regions and require user specification and/or ad hoc metrics to select the sensitive data to protect.

In this paper, we propose the first approach that fully protects ***safe*** stack objects from attacks on memory errors efficiently. To do so, we develop a safety analysis that identifies the stack objects that are safe from memory errors comprehensively, enabling their protection from references to other (potentially unsafe) objects via isolation on a separate stack. To ensure security, our proposed analysis is conservatively designed to either prove that all accesses to a stack object must be safe or classify that object as unsafe (i.e., may not be safe). We focus on stack objects because they typically have simpler memory layouts, use constant allocation frequently, and have more clearly defined scopes (e.g., deallocated on function returns), which increase the likelihood of successful safety validation. While our goal may have little impact on attacks on memory errors on heap and global objects, stack memory errors are still common and often provide powerful exploit opportunities. Providing a foundation that isolates all the stack objects for which safety can be proven systematically protects many objects that may otherwise be prone to stack memory exploits, and does so for practical overheads.

Identifying safe stack objects to protect them effectively and efficiently presents several challenges. First, there is the challenge of ensuring that safe stack objects are safe from memory errors comprehensively. In order for stack data and pointers to be protected from exploitation without runtime checks, we must validate safety for ***all*** classes of memory errors: spatial, type, and temporal. If we only prove that a stack object is safe from buffer overflows (one type of spatial error), type and temporal errors may still enable an adversary to maliciously modify or disclose other stack objects. Second, there is the challenge of increasing the number of stack objects protected from memory errors statically. We must devise techniques to validate safety requirements that are more accurate, yet are still practical to apply across entire codebases. Third, there is the challenge of ensuring that the safety validation does not misclassify an unsafe object as safe. The challenge is to ensure that all stack objects that are proven to be safe from memory errors are actually safe.

We present the DATAGUARD system, which aims to address these challenges to protect a large fraction of stack objects from exploits on memory errors efficiently. First, DATAGUARD determines whether a stack object is safe from spatial, type, and temporal memory errors by validating *memory safety constraints* generated automatically for each stack object. Only if a stack object is proven safe from these three types of memory errors can it be placed on an isolated stack to ensure its integrity without runtime checks. Second, DATAGUARD identifies a greater number of stack objects as safe than prior techniques by augmenting static analyses with a targeted symbolic execution to remove false positives (i.e., falsely unsafe cases). The symbolic execution leverages the static analysis to only evaluate program paths that may cause an unsafe operation. Third, all the static analyses and symbolic execution methods in DATAGUARD are designed to prevent the misclassification of any unsafe stack objects as safe by configuring and combining the multiple analyses necessary to achieve the desired accuracy in a manner that terminates (i.e., classifies an object as unsafe) whenever safety can no longer be guaranteed. DATAGUARD then applies the Clang Safe Stack defense unmodified, enabling the protection of more stack objects from memory errors comprehensively. An additional benefit of the DATAGUARD protection is that by finding a greater number of safe objects, in particular a greater number of functions with only safe objects, the performance overhead of stack protection is reduced.

We evaluate DATAGUARD on nginx, httpd, proftpd, openvpn, and opensshd servers and the SPEC CPU2006 [36] benchmark programs, finding that DATAGUARD improves the accuracy of classification to improve security and performance. First, DATAGUARD avoids misclassifying a significant fraction of unsafe stack objects as safe. For example, over 60% of the cases found unsafe for type errors and 6.3% of all stack objects found unsafe by DATAGUARD are classified as safe by Safe Stack[2]. Second, DATAGUARD is able to prove safety from spatial, type, and temporal errors for 91.45% of stack objects on average for these programs. As a result, over 70% of the stack objects classified as unsafe by Safe Stack are classified as safe by DATAGUARD, enabling DATAGUARD to protect 18% more stack objects on average. We examine the security impact of DATAGUARD's classification on the CGC Binaries [23], finding that, for the 87 binaries that include at least one stack-based vulnerability, all exploits are thwarted by DATAGUARD's classification by isolating safe objects. 95 of the 118 vulnerabilities are thwarted directly, and the other 23 cases are thwarted indirectly because they must exploit a second memory error to complete their attack. Finally, experiments show that DATAGUARD can leverage Clang's Safe Stack runtime defense, where the DATAGUARD safety validation results in a performance improvement across the SPEC CPU2006 benchmarks from 11.3% for the Safe Stack classification to 4.3% for DATAGUARD.

This paper makes the following contributions:

- *We address the goal of marking as many stack objects "safe" as possible* while ensuring that *no unsafe stack object is ever classified as safe* relative to spatial, type, and memory errors.
- *We propose the* DATAGUARD *system - a significantly more accurate method for validating the safety of stack objects* against spatial, type, and temporal memory errors to increase the scope of protection for stack objects.
- *We provide a novel set of stack object safety constraints* and develop safety analysis for three classes of memory errors that

[2]As Safe Stack focuses on protecting code pointers only, this limitation does not create significant attack vectors, but is insufficient for protecting stack data objects that are prone to attacks on these memory errors.

```
1 void example(int ct, char **buf) {
2    int lct = BUF_SIZE;
3    char lbuf[lct];
4    if (ct < lct){              //(1) ct > buf's size
5      strlcpy(lbuf, *buf, (size_t) ct);    //(2) ct < 0
6    }
7    *buf = lbuf;                //(3) temporal
8 }
```

Fig. 1: Example function demonstrates: (1) bounds error that enables overread of `buf`; (2) type error due to casting of `ct` from signed to unsigned; and (3) temporal error as `*buf` references local variable `lbuf` after return.

combines static analysis and symbolic execution to maximize the number of proven safe stack objects, validating the safety over 65% of the stack objects found unsafe by Safe Stack.

• *We find that* DataGuard *improves the security of safe stack objects* by removing 6.3% of stack objects misclassified as safe by the Safe Stack due to its incomplete protection against memory errors and *prevents the exploitation of stack memory errors* in CGC Binaries for only 4.3% performance overhead on average for the 16 supported SPEC CPU2006 benchmarks.

II. Motivation

In this section, we motivate the need to protect stack objects from memory safety violations and show that current defenses are too limited and/or expensive.

A. *Exploiting Memory Errors on Stack Objects*

Figure 1 shows the function `example`, which demonstrates the three classes of memory errors that we examine in this work. Assume that the value assigned at `ct` may be controlled by an adversary.

First, line 4 demonstrates a *spatial error*, which permits accesses (i.e., reads or writes) outside the memory region of a stack object. In this case, the spatial error occurs because the value of the size parameter `ct` may be larger than the actual size of the memory region allocated for `*buf`, whose own size could also be smaller than the defined constant `BUF_SIZE`. Thus, an adversary could read the memory objects following `*buf` in the stack segment to exfiltrate sensitive data from other stack objects. In general, spatial errors may enable access to memory prior the buffer (i.e., underflows) as well.

Second, line 5 demonstrates one form of a *type error*, which causes a stack object to be interpreted in unexpected ways. In this case, a type error occurs because the value `ct` may be negative (as a signed integer), but is cast to positive value (as `size_t` is unsigned) at line 5, converting a negative value to a large positive value that also causes a bounds error (i.e., a buffer overflow) that may modify sensitive stack data.

Third, line 7 demonstrates one form of a *temporal error*, which permits access to a memory object that has been deallocated. In this case, the pointer `*buf` is assigned to the memory location referenced by the local pointer `lbuf`. Since `*buf` may be used after the function `example` returns (e.g., in the function that calls `example`), this assignment allows those uses to reference memory that is out of scope, creating a *dangling pointer*. Temporal errors may also cause use prior to initialization as well as or use after deallocation. In addition, temporal errors on uninitialized data may cause memory errors if the data is used to compute memory references.

All three classes of memory errors on stack objects are still frequently discovered. Recent critical vulnerabilities (i.e., a CVSS 3.x severity base score of over 7.5) include those for spatial errors (e.g., CVE-2021-25178, CVE-2021-3444, CVE-2020-25624), type errors (e.g., CVE-2021-26825, CVE-2020-15202, CVE-2020-14147), and temporal errors (e.g., CVE-2020-25578, CVE-2020-20739, CVE-2020-13899). While historically stack exploits have often targeted code pointers (e.g., return addresses), the need to circumvent stack defenses (see Section II-D) has motivated other attack vectors, such as the modification of control data (e.g., line 6 of Figure 1) and exfiltration of sensitive stack data (e.g., line 4 in Figure 1). We examine how DataGuard prevents an exfiltration attack (CVE-2020-20739) in Section VII-G. As a result, defenses that protect stack objects from all three classes of memory errors systematically are a necessary foundation for software security.

B. *Current Defenses*

A set of stack defenses were proposed to prevent exploits that modify return addresses, such as *Stack Canaries* [20] and *Shadow Stacks* [15]. These defenses can now be implemented reasonably efficiently ($< 5\%$ overhead [4], [11], [92]), but stack objects other than return addresses are also prone to attack. Given that advanced adversaries can launch successful attacks by modifying non-return-address stack objects to redirect control flow (e.g., non-return-address code pointers or data used in control-flow decisions) and to exfiltrate sensitive information, limited stack defenses are now insufficient.

Researchers have long recognized this gap and proposed runtime defenses to prevent an entire class of memory errors comprehensively, such as to enforce spatial safety [3], [28], [60], [89], prevent attacks on type errors [35], [41], [47], and prevent temporal safety violations [25], [44], [46], [88], [91]. but these defenses individually have significant overheads, even when applied only to stack objects [28] in some cases. So researchers have proposed optimizations to remove some runtime checks for references that cannot violate bounds [3], [24] or can never become dangling references [26], [27]. An issue is that the underlying static analysis techniques under-approximate the number of truly safe objects to avoid misclassifying unsafe objects as safe, but may miss a significant fraction of truly safe objects. Ultimately, we want defenses to protect as many stack objects from these classes of attacks as possible in reasonable overhead.

An alternative approach focuses on protecting objects that can be proven safe from memory errors without runtime checks. For stack objects, such protection can be provided by using multiple stacks [90], where each with stack objects satisfying distinct requirements. The *Safe Stack* defense [45] applies the multistack [90] approach to protect stack objects by separating objects whose references are determined safe by the compiler onto a “safe” stack isolated from other “unsafe” objects on the “regular” stack. While the focus of the Safe Stack defense is to protect code pointers (i.e., in addition to the return addresses), it also protects other stack objects found to meet its safety criteria, resulting in the ability to protect over 60% of stack objects for the programs assessed in Section VII without runtime checks. We examine how the safe defense works and limitations next to motivate the need of providing a more secure, effective, and efficient defense.

C. Safe Stack Background

The safe stack defense consists of a static analysis pass to classify safe/unsafe stack objects, an instrumentation pass to place and reference stack objects on their respective stacks, and runtime support to ensure the integrity of the safe stack. The static analysis pass classifies objects as safe if they are only accessed using a constant (i.e., compiler-determined) offset from the stack pointer within a single stack frame. While code pointers, such as the return addresses, often satisfy this requirement, some stack data objects and data pointers may also comply. The instrumentation pass creates two separate stacks to separate safe objects from unsafe objects. The runtime support protects the safe stack by allowing only accesses to the safe stack through authorized instructions via dedicated registers, such that no addresses of the safe stack nor pointers that point to the safe stack are ever stored on the regular stack, preventing the corruption of safe objects from tampering with unsafe objects.

To demonstrate how the Safe Stack approach works, return to the function `example` in Figure 1. In this example, the Safe Stack approach classifies `lct` as safe, placing it on the safe stack, and `lbuf` as unsafe, leaving it on the regular (unsafe) stack. The variable `lct` is only accessed using a constant offset from the stack pointer. On the other hand, `lbuf` is passed as a parameter to `strlcpy` in line 5, so its accesses in `strlcpy` will not be a constant offset from the stack pointer in that function. In the latter case, a bug in the program could change the memory reference to `lbuf` from something other than what the compiler defined, and such objects are classified as unsafe by the Safe Stack approach. Note that the pointer operations for scanning `lbuf` in `strlcpy` are also considered unsafe because they do not use a constant offset either.

D. Limitations of the Safe Stack Defense

While the *Safe Stack* defense [45] applies the multi-stack [90] approach to eliminate runtime checks by isolating the safe stack, Safe Stack does not validate safety for all three classes of memory errors, making it insufficient to protect data objects in general. In particular, Safe Stack does not account for type errors at all nor use-before-initialization temporal errors, which permits adversaries to violate memory safety on stack data objects and their references. To be fair, the focus of Safe Stack is on protecting code pointers, rather than stack objects in general. For code pointers, type errors do not impact return addresses and other safe (to Safe Stack) code pointer variables, and exploiting code pointer variables via use-before-initialization is rare and difficult. However, type and temporal errors are often exploited to enable attacks on stack data.

In addition, Safe Stack's safety requirements are overly conservative, leaving many objects unprotected unnecessarily. For example, Safe Stack is particularly conservative because it declares all address-taken variables as unsafe, which includes all arguments passed by reference. We find that many of these objects can be proven safe even when accounting for all three classes of memory errors.

In summary, existing multistack defenses, like Safe Stack, protect some stack objects without runtime checks, but do not protect stack objects from all three classes of memory errors systematically, which leads to false negatives that risk the integrity of stack protection. In addition, many objects that could be proven safe are not at present due to false positives resulting from overly conservative invariants, which both risks the security of these objects and degrades the performance.

```
1  int callee(int ict, char *ibuf, int *oct, char *obuf ) {
2    if (ict < *oct){
3      if ((strlcpy(obuf, ibuf, (size_t)ict)) >= *oct)
4        return -1; // Truncation Check
5      *oct = (size_t) ict;
6    }
7    else{
8      if ((strlcpy(obuf, ibuf, (size_t) *oct)) >= *oct)
9        return -1; //Truncation Check
10   }
11   return 0;
12 }
13
14 int caller(int fd, char *in) {
15   int imax = 100, omax = imax-10;
16   char ibuf[imax], obuf[omax];
17
18   if ((strlcpy(ibuf, in, (size_t)imax)) >= imax)
19     goto error;  // Truncation Check
20   if ((callee(imax, ibuf, &omax, obuf)) < 0)
21     goto error;
22   if ((write(fd, obuf, (size_t)omax)) == omax)
23     return 0;
24 error:
25   return -1;
26 }
```

Fig. 2: Revised example where the stack objects are provably safe from bounds, type, and temporal errors.

III. Overview

The core challenge of this work is to identify a maximal number of stack objects that are safe from spatial, type, and temporal memory errors without misclassifying any unsafe stack objects as safe. These safe stack objects are then isolated to protect them from possibly unsafe memory accesses to other objects. To achieve this goal, we propose the DataGuard system that implements the approach shown in Figure 3. DataGuard performs a multi-step analysis that combines static analyses with constrained symbolic execution to validate stack object safety to prevent unsafe stack objects from being misclassified as safe. Once the safe stack objects are validated, DataGuard applies the Safe Stack defense of isolating safe stack objects to protect them at runtime.

To demonstrate the goals of DataGuard, Figure 2 shows an example of code where the safety of stack objects can be validated statically. The function `caller` allocates buffer sizes (i.e., `imax` and `omax`) as constants to allocate buffers of constant size (i.e., `ibuf` and `obuf`). A key limitation of the Safe Stack approach [45] is that all stack objects passed as parameters are classified as unsafe. While the caller in Figure 2 passes the constants and buffers to the callee, none of these parameters have accesses that could cause spatial, type, or temporal errors. First, although the caller uses a signed integer type for the buffer sizes and these are cast in the callee (lines 3 and 8), the constant values are known and will be unchanged by the cast, so no type error is possible. Second, although the buffers are used in copy operations (lines 3 and 8), the copies are bound by the `strlcpy` function to be within the bounds[3]Security issues caused by truncation are not in the scope of this paper. (as determined by the constant

[3]`strlcpy` also guarantees the resulting string is null-terminated and enables detection of truncation.

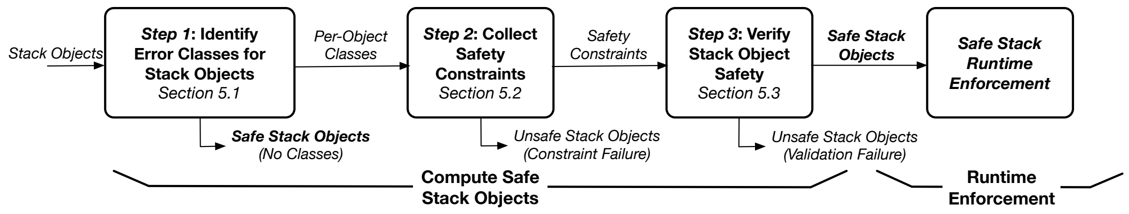

Stack Objects
Step 1: Identify Error Classes for Stack Objects Section 5.1
Per-Object Classes
Step 2: Collect Safety Constraints Section 5.2
Safety Constraints
Step 3: Verify Stack Object Safety Section 5.3
Safe Stack Objects
Safe Stack Runtime Enforcement
Safe Stack Objects (No Classes)
Unsafe Stack Objects (Constraint Failure)
Unsafe Stack Objects (Validation Failure)
Compute Safe Stack Objects
Runtime Enforcement

Fig. 3: DataGuard system technical approach

buffer sizes with safe type casts). Regarding temporal safety, the buffer references are initialized, their values are initialized prior to dereferencing (we assume `in` is initialized also), and no memory in `callee` is assigned to these references (i.e., in fact these references are unchanged). Thus, the buffers and their sizes can be isolated of a separate stack (i.e., *isolated stack*) without risking other safe stack objects to protect them from memory errors in other possibly unsafe stack objects (i.e., on the *regular stack*) and other program objects.

DataGuard validates stack object safety from memory errors in three steps, as shown in Figure 3. First, DataGuard identifies the classes of memory errors that may apply for each stack object, as described in Section V-A. Each class requires that the stack object be accessed via an associated, unsafe pointer operation. A stack object is declared *safe* if no pointer that may reference the object is used in an unsafe operation. DataGuard leverages prior work for identifying the pointer operations that may cause spatial and type errors [16], [61], [62] and proposes how to detect pointer operations required to cause temporal errors. Second, DataGuard generates constraints for validating the safety of stack objects for each class of memory error automatically, which we call *memory safety constraints*, as described in Section V-B. A stack object is declared to be *unsafe* should DataGuard not be able to generate safety constraints for a class requiring validation. Third, DataGuard validates each remaining stack object's safety using its safety constraints, as described in Section V-C. DataGuard applies a static analysis first, where the stack objects that pass validation are declared *safe*. For the objects not found to be safe statically, DataGuard applies a second analysis; this time by employing targeted symbolic execution to validate safety. To ensure that we do not misclassify any unsafe objects as safe in either analysis, we require that both analyses overapproximate the possible executions of the program, i.e., are *sound* analyses[4]. The stack objects that pass either validation are declared *safe*, whereas others are *unsafe* (i.e., cannot be proven safe in either analysis). We assess the soundness of our approach in Section V-D.

DataGuard uses existing runtime enforcement, Clang's Safe Stack, to protect the safe stack objects identified by DataGuard. DataGuard both removes a significant number of objects that may be unsafe that other techniques classify as safe to protect the integrity of the isolated stack and finds a much greater number of safe objects that prior techniques classified as unsafe, which extends stack protections to more objects and reduces the overhead of the Safe Stack defense. Thus, DataGuard creates a win-win situation by extending stack protections while reducing performance overhead.

[4]Computer science communities have differing viewpoints of soundness and completeness. We are using the static analysis community's definition, as in [74], where a sound analysis overapproximates the program's executions.

IV. Threat Model

In this section, we list threats to systems on which DataGuard will be deployed and outline trust assumptions upon which DataGuard depends in thwarting those threats.

We assume each program protected using DataGuard is benign but may contain memory safety errors, including spatial, type, or temporal errors. We further assume that adversaries will exploit memory errors on any stack object in a program. As described in Section II-A, adversaries can exploit such memory errors in a variety of ways. We assume stack objects classified as unsafe by DataGuard are prone to such memory errors and attacks, as are heap and global data. We leave the problem of extending protection to unsafe stack objects for future work.

We assume that no stack protection mechanism is deployed except for ASLR, as DataGuard proposes a mechanism to supplant and extend prior stack defenses. DataGuard uses ASLR to isolate the isolated stack from other references by placing the isolated stack in an unpredictable location, as used in Clang's Safe Stack defense. We note that this leaves the attacker a (small) probabilistic window of successfully compromising an object on the isolated stack through information disclosure, which is mainly triggered by two approaches: (1) taking advantage of implementation flaws [19], [29] and (2) performing just-in-time information disclosure attack after program load time to decrease the entropy [33], [34], [63]. As an alternative to ASLR, the isolated stack can be guarded from other memory accesses through SFI [86] or hardware isolation, such as Intel MPK [59], [66], [85] for some additional overhead. Existing works on enhancing the information-hiding property [11], [87], preventing information leaks [55], and enforcing access control on sensitive data [31], [37], [43] can also be applied to strengthen the security of Safe Stack. We assume that control-flow integrity [1] (CFI) is deployed and that the program is not permitted to modify its own code, i.e., the code memory is not writable and data memory is not executable to ensure that our static and symbolic analyses really examine an overapproximation of the possible program executions. We also assume that all constant values are stored in read-only memory and are not copied into read-write memory at any time [32].

V. Design

We review the design of the DataGuard system, which comprises the first three steps of Figure 3, and present soundness arguments for the proposed safety validation analyses.

A. Identifying Error Classes for Stack Objects

Researchers do not yet have a broadly accepted definition of safety from memory errors. Distinct safety definitions were utilized by each of the Safe Stack defense [45], CCured [16], [61], [62], and SAFECode [24], [26], [27]. In this paper, our definition of safety is closest in spirit to the CCured system, where each object is strongly typed and a safe object complies with its spatial (bounds) and type semantics for all program executions. A limitation of the CCured approach is that it does not provide a safety definition for temporal errors, so we rely on the SAFECode definition for temporal errors. SAFECode defines safety in terms of points-to information, where an object is safe if it obeys all its computed points-to relationships, which enables SAFECode to detect dangling and uninitialized pointers as points-to violations.

• **Definition 1**: A stack object is *safe* if any references (pointers) that may alias to the object comply with the spatial, type, and temporal safety requirements associated with the object and the references themselves are safe objects.

As Definition 1 states, for a stack object to be safe from memory errors, all pointers that may-alias the object must be comply with safety requirements for the stack object. Thus, DATAGUARD computes the pointers that may-alias a stack object and collects the classes of memory errors associated with those pointers. If none of a stack object's pointers performs any operation that may cause a memory error, that stack object is safe.

The first step that DATAGUARD takes to determine whether a stack object is safe, as shown as Step 1 in Figure 3, is to identify the classes of memory errors for each stack object that require safety validation. To do this, DATAGUARD leverages the techniques proposed by CCured (spatial and type safety) and SAFECode (temporal safety) to determine whether a pointer may be used in a memory operation that may cause a particular class of memory error. The classes of memory errors that must be verified for a stack object are the union of the classes that may be caused by its aliases.

CCured identifies pointers that may cause spatial and type errors. First, CCured shows that pointers used in pointer arithmetic operations (called sequential or *seq* pointers by CCured), such as to access array elements or structure fields, may cause spatial memory errors. While CCured requires runtime checks for such pointers, DATAGUARD labels any objects that may be referenced by such pointers as requiring validation for the spatial error class. Note that DATAGUARD treats fields within compound objects, including buffers, as distinct variables, as described in Appendix C.

Second, CCured shows that pointers used in type casting operations may cause type errors. CCured does aim to prove safety for upcasts and downcasts [16], but these casts are uncommon for stack objects. DATAGUARD labels objects that may be referenced by such pointers as requiring validation for the type error class. Note that LLVM casts unions into structures or primitive types, and it occasionally use *ptrtoint* and *inttoptr* cast instructions for accessing fields of compound objects; so such cases are identified by DATAGUARD as requiring type safety validation. However, because type errors are uncommon for stack objects (see Table II) and largely ad hoc, we only validate unsigned-signed casting for integers of the same allocated size to identify integers whose values are not impacted by type cast, which for DATAGUARD to expand the number of cases that can pass spatial safety validation.

To detect operations that may lead to temporal errors for pointers, DATAGUARD identifies dangling pointers. The SAFECode system [26], [27] includes a method to validate stack objects that are prone to dangling pointers based on *escape analysis*. Rather than running full escape analysis, DATAGUARD detects pointers that may escape to calling functions or other threads (e.g., via heap or globals) based on their aliases. If any pointer is aliased by a pointer passed from a caller or that references a heap or global object, then DATAGUARD identifies the need for temporal analysis.

DATAGUARD also identifies stack objects that may cause temporal errors because they may be used prior to initialization and hence reference stale memory. In this case, both pointer and data variables may be prone to use-before-initialization. If a variable is initialized to a value at declaration, it does not require temporal analysis for UBI, otherwise it does.

B. Collecting Stack Object Constraints

DATAGUARD uses the following approach to collect constraints. First, for each stack object, DATAGUARD collects constraints from the stack object's declaration for the classes of memory errors that require safety validation. These constraints are assigned to pointers whenever that stack object is assigned to a pointer (e.g., at pointer definitions). For static analysis, we make such assignments based on whether the pointer may alias the stack object. For symbolic execution, we make such assignments when the symbolic execution finds that the stack object may be referenced by the pointer. Should multiple stack objects be assigned to the same pointer (i.e., at the same pointer definition with an LLVM phi instruction), the analyses proceed independently, as these represent distinct contexts. State relevant to safety validation (e.g., index of a pointer in a buffer) may be updated at each pointer definition and each pointer operation (i.e., at pointer use). Constraints are then validated on each pointer use.

Spatial Constraints: As is typical since the CCured method was proposed [61], checking spatial safety involves determining the size of the object and ensuring that all accesses are within the bounds determined by that size. To check such a requirement statically, we require the following information. Stack objects that do not satisfy the following four constraints will remain on the regular, unprotected stack.

• ***Declaration***: The *size* from the object's *base* must be declared as a constant value. The initial *index* is 0.

• ***Definition***: When a pointer is defined to reference the object, the reference may be *offset* to change the index. This offset must be a constant value.

• ***Use***: When a pointer is used in an operation, the pointer may be further offset to change the index. Each offset in a use must also be a constant value.

• ***Validation***: For all uses, pointer *index* $<$ *size* and *index* ≥ 0.

To check that a stack object is free of spatial errors, the stack object must be declared with constant size. An index of the pointer into the stack object is maintained. Pointer definitions and uses may change the value of the index by a constant offset. The resultant index from the combination of offsets in pointer definitions and uses must be within the range of the stack object on each use, i.e., greater than 0 and less than object size.

Type Constraints: The use of structured types and type casting among them is less common for stack objects, so DATAGUARD focuses on validating the safety of integer type casts. Below, we describe the four requirements for type safety validation for integer stack objects and the definition and use of their references.

• ***Declaration***: Integer variables are assigned the *type* and *value* (optionally) used in the declaration.

• ***Definition***: When a pointer is defined to reference the object, if the operation includes a type cast the *newtype* is identified. If validation succeeds *type* is assigned to the *newtype*.

• ***Use***: When a pointer is used to reference the object, if the operation includes a type cast the *newtype* is identified. If validation succeeds *type* is assigned to the *newtype*. If the operation assigns a value, the *value* is stored.

• ***Validation***: For a definition or use that produces a *newtype*, the resultant type cast must not change the *value* of the integer object referenced (e.g., by changing size or signedness).

For type safety validation, we require that type casts not change the value associated with the reference to change. This check enables validation of safety from integer overflows, e.g., to prevent attacks on control data.

Temporal Constraints: For temporal constraints, we focus on scoping constraints to prevent (1) memory use before it is initialized (i.e., use of aliases before the stack object is declared), and (2) memory use after it is deallocated (i.e., upon return of a function in which the stack object is declared).

• ***Declaration***: The object is declared in basic block $b_{obj_{init}}$ implying that it may be live in a set of basic blocks B_{obj}.

• ***Definition***: The pointer is defined in basic block $b_{ptr_{def}}$.

• ***Use***: The pointer uses occur in basic blocks $B_{ptr_{use}}$.

• ***Validation***: The pointer definition occurs when the object is live, $b_{ptr_{def}} \in B_{obj}$, and all uses occur in a basic block in where the object is still live, $B_{ptr_{use}} \setminus B_{obj} = \emptyset$.

An object cannot be used prior to initialization as long as there cannot be a use of the alias before the declaration of the object. An object cannot be referenced by a dangling pointer as long as there cannot be a use of the object after its lifetime, e.g., after the function in which it was declared returns. We define this constraint in terms of the basic blocks in which an object may be live given the object's declaration. Then, DATAGUARD validates that the pointer definition occurs when the stack object is live and that all uses occur in a basic block in which the stack object is still live.

C. Validating Stack Object Safety Statically

In this section, we specify methods to validate stack object safety with respect to the constraints from the last section. The challenge is to maximize the number of stack objects that are safe without misclassifying any unsafe stack objects as safe. We first apply a two-stage analysis that uses an inexpensive static analysis first, followed by symbolic execution that leverages the relevant program paths found by static analysis. Note that researchers have previously used symbolic execution to determine whether positives found in static analysis are actually true positives (e.g., a recent example is the UbiTect system [91]). However, our two-stage analysis for safety validation differs from bug finding analyses in that each positive must be a true positive for all possible program executions, requiring that all the static and symbolic analyses used in safety validation must overapproximate all program executions, i.e., be *sound* analyses.

Validating Spatial Safety. Spatial safety analysis is performed in two stages: (1) a static analysis to classify pointers to find pointers that may be proven safe, similarly to the Baggy Bounds [3] and SAFECode [24], [25] optimizations, and (2) a guided symbolic execution to determine whether the pointers found to be unsafe in the first stage can be proven safe in a more comprehensive analysis (i.e., are false positives).

For the static analysis, DATAGUARD first performs a *def-use analysis* [52], [84] on each pointer detected to require spatial safety validation. LLVM uses SSA form for its intermediate representation, so each pointer has only one definition, but may have many uses. The computed def-use chains are used in both the static analysis and to guide the symbolic execution. To assess bounds statically, DATAGUARD performs a *value range analysis* [8], [72], [81] for each pointer use from its definition. Value range analysis is a type of data-flow analysis that tracks the range (interval) of values that a numeric variable can assume at each point of a program's execution. To validate spatial safety for all possible program executions, every pointer that may alias the stack object must use offsets from the object's base whose value range is between 0 and stack object size as determined by the spatial safety constraints.

A problem is that traditional value range analysis assumes that values in memory are not prone to memory errors, so the value ranges computed may not be correct under such errors. Baggy Bounds [3] does not specifically mention this problem, although since Baggy Bounds aims to prevent all spatial errors through runtime checks, this assumption is not necessarily inconsistent for that method. For DATAGUARD, unsafe stack objects may remain and we cannot trust that the data in these stack objects are protected from memory errors. Instead, our value range analysis only uses constant values, either from def/use instructions explicitly or from loads of constant values into registers used by them. All other cases are identified as unsafe. Thus, under the assumption that code and constant values are loaded in read-only memory (see Section IV), spatial safety validation is not impacted by those memory errors.

For the cases that the static analysis finds to be unsafe, a problem is that these cases may be false positives due to the overapproximations inherent in the sound analysis. DATAGUARD employs a second stage that performs a guided symbolic execution to determine if it can prove whether any of the positives found in static analysis can actually be proven safe. To do this, DATAGUARD symbolically executes the part of the program covered by the def-use chains of each pointer that may alias the stack object, starting from the stack object declaration. To satisfy the requirement of using constant inputs for sizes and offsets, the symbolic execution relies on the value range analysis. If a stack object is inferred to be unsafe by value range analysis, it either does not use a constant input or is detected to violate bounds. The former is more likely, so DATAGUARD classifies pointers that rely on such data as unsafe, only analyzing the latter cases. We find that imprecision that may cause false positives occurs due to the lack of path sensitivity, where the effects of multiple program branches may falsely indicate that bounds are violated. Symbolic execution can check these paths independently. All non-constant data remain symbolic, and the use of symbolic data in computing sizes or offsets causes a failure in the safety validation.

A challenge in using symbolic execution is to avoid path explosion. Since DATAGUARD employs symbolic execution on the positive (i.e., presumed unsafe) cases found by static analysis, we only need to symbolically execute the program from the stack object's declaration to its last unsafe use in its def-use chains, which improves DATAGUARD's ability to identify safe cases over pure symbolic execution without static analysis based on our evaluation on Section VII-D. DATAGUARD applies LLVM's loop simplify [54] and canonicalization technique [53] to reduce the path explosion problem introduced by loops, as applied in patch generation to satisfy safety properties [39]. DATAGUARD also leverages symbolic state merging employed by S2E [14] to eliminate unnecessary forks when encountering branches, which reduces the number of paths to explore by orders of magnitude [70]. To avoid path explosion in long sequences of def-use chains, we limit the analysis depth. As longer def-use chains are more likely to lead to unsafe operations, we focus on shorter paths.

Validating Type Safety. Since validating type safety involves checking that values do not change on type casts, the core of type safety validation is in the static value range analysis. If a stack object that is validated for type safety is also used as a bound or memory (e.g., array) index, DATAGUARD uses the result of type safety to validate spatial safety (i.e., all such integers must be safe from type errors).

Validating Temporal Safety. The intuition of the proposed approach to detect dangling pointers is to determine whether a pointer use may occur outside the scope of a referenced stack object. To detect such cases without misclassifying any unsafe cases as safe, DATAGUARD uses a sound *liveness analysis* [5], [50], [51] to determine the basic blocks in which a stack object is in scope in the program (its live set) and determine whether any uses of pointers that may reference the stack variable occur in basic blocks outside the live set. However, false positives may occur in aliasing, so we validate that the pointer can actually be assigned to the stack object using symbolic execution for any positive cases.

Liveness analysis computes the variables that are alive at each basic block, so we can determine the set of basic blocks in which a stack object declared in block $b_{obj_{init}}$ is live as B_{obj}, which is called the *live range* [9], [10], [49] of obj. Similarly, we compute the live ranges for each of the pointers that may-alias obj, which determines the set B_{ptr}. Since LLVM uses SSA form for its IR, DATAGUARD evaluates safety for a stack object relative to each pointer individually. To be specific, the live-range of the pointer ptr that may alias an object obj begins at the basic block when ptr is declared, and it ends after the last use of ptr. For each pointer ptr and aliased object obj, DATAGUARD validates whether any use of ptr (in ptr's live-range) is outside the obj's live-range. We note that aliases may have uses before the stack object is declared or after the stack object's live-range has completed. In static analysis, either case results in the stack object being classified as unsafe.

However, we also recognize that if a pointer is found unsafe by the live-range analysis, it may not actually be exploitable, since overapproximation in alias analysis may identify aliases that cannot actually reference the stack object. As a result, we similarly propose to apply guided symbolic execution to validate whether each stack object found to be unsafe is really assigned to a pointer whose uses may occur outside the stack object's live range. If the symbolic execution proves that such a reference is not possible for all pointers failing temporal validation statically, then the stack object is found to be safe. To do that, we again track the computed def-use chains. For each stack object, we start the symbolic execution at the object's declaration and follow all the def-use chains for all the aliases, similarly to symbolic spatial validation, including the methods to avoid path explosion.

D. Validation Soundness

In this section, we assess the soundness of the proposed static analyses and symbolic execution for safety validation.

Static Analysis Soundness The static analyses methods for safety validation are built upon multiple prior static analyses. They include: (1) LLVM's built-in def-use analysis; (2) SVF pointer alias analysis using its VFG [76]–[80]; (3) Program Dependence Graph (PDG) of PtrSplit [48]; (4) value range analysis [8], [72], [81]; and (5) live range analysis [9], [10], [49]. These static analyses are claimed to be sound by their respective papers. However, we note that the soundness arguments in these papers are informal and there are no formal soundness proofs (with the exception of value range analysis, whose soundness proof is formalized [72]). Therefore, it is possible that these analyses and their implementations do not handle all corner cases in sound ways. To separate the analyses DATAGUARD relies on from how DATAGUARD applies them, we next argue that DATAGUARD's static analyses achieve *relative soundness*: assuming those prior analyses are sound, DATAGUARD's static analyses are sound. We note that DATAGUARD avoids several corner cases (e.g., type casting) by design by classifying objects accessed using such operations as unsafe, as listed in Appendix C.

DATAGUARD's safety validation methods require finding all pointers that may alias each stack object. For that, DATAGUARD relies on the SVF pointer alias analysis [76]–[80] intraprocedurally based on its VFG and the PDG representation of PtrSplit [48] to represent the data flows between functions to compute interprocedural may-aliases. Since both the VFG construction in SVF and the PDG representation are claimed to be sound, DATAGUARD soundly overapproximates the set of pointers that may alias a stack object.

The spatial and type safety validations apply value range analysis [8], [72], [81] to compute the possible ranges of indices in accessing stack objects. The value range analysis is computed based on the PDG and tracks only simple patterns: (1) pointers with constant offsets from bases of stack objects and (2) constant offsets when dereferencing pointers. Since the PDG is claimed to be sound and we assume (Section IV) that all constants are stored in read-only memory and are not copied into read-write memory at any time [32], it is straightforward that DataGuard's value range analysis is sound.

The temporal validation applies liveness analysis [50] to compare the basic blocks in which a stack object is live to those basic blocks of each pointer that may alias the object. Liveness analysis is also computed based on the PDG and tracks the def-use chains and object's scope in a fully context-sensitive manner. Since the PDG is claimed to be sound, it is straightforward that DataGuard's liveness analysis is sound.

Symbolic Execution Soundness By default, symbolic execution is a sound form of analysis because it follows all execution paths in a program [7]. In practice, factors complicate ensuring the soundness of a particular symbolic execution analysis that DataGuard addresses.

First, path explosion in symbolic execution often means that it is impractical to execute all paths in the program, even with loop canonicalization and symbolic state merging. DataGuard limits the depth of the symbolic execution to avoid expensive cases, but any symbolic execution terminated for this reason classifies the associated stack object as unsafe.

Second, symbolic analyses may sacrifice soundness when employing concrete values for some variables, creating a concolic execution [7], [14]. DataGuard only concretizes values that are constants in spatial safety analysis. All other variables are initialized with symbolic values. Only symbolic values are used to initialize variables in the temporal analysis.

Third, the symbolic analyses in this paper do not start at the program initialization, so if other execution contexts (i.e., threads) have been created they may impact the state of the symbolic execution. DataGuard validates whether a single stack object (at a time) is safe relative to spatial and temporal errors. This would imply, if successful, the object would be on an isolated stack, protected from tampering by accesses from this thread or others to unsafe globals, stack and heap objects.

Fourth, DataGuard's symbolic executions utilize the def-use chains soundly in validating stack objects found unsafe by static analysis. The symbolic execution analysis starts at the stack object's declaration and symbolically executes the program until the last unsafe pointer operation (use) of any alias of the stack object. Thus, all paths that can possibly lead to a memory error are executed symbolically before a stack object can be declared safe. DataGuard only allows safe objects (i.e., proven by the static analysis) to be used in any constraints derived from the symbolic execution. Unsafe objects remain symbolic, so they are not constrained during the execution. The loop-canonicalization applied to simplify loops [39] and symbolic state merging [70] are also sound.

VI. Implementation

DataGuard is implemented 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 64 GB RAM. The CCured pointer analysis tool is ported, adapted, and extended from nesCheck [57] which consists of 1,958 SLoC in C++. Originally, nesCheck aims to analyze TinyOS where all code for applications, libraries, and OS is fully available at compile time, so that the whole-program static analysis can be done effectively. In our work, we statically link libraries at compilation to make the code of the library available in bitcode. Also, we found that the original nesCheck framework fails to propagate changes in the classification of global pointers. To solve this problem, we port the PDG of the PtrSplit framework [48] which leverages SVF's [77] alias analysis into the framework to propagate classifications to pointers comprehensively. The static analysis portion of DataGuard consists of 4,568 SLoC in C and C++.

The def-use chain implemented in DataGuard is derived from LLVM's built-in def-use analysis. The interprocedural alias analysis is based on the PDG-based alias analysis from PtrSplit [48], using the SVF analysis for intra-procedural pointer alias analysis. The implementation of value range analysis, live range analysis is based on the data-flow equations provided in Appendix A and B. For the symbolic execution, we apply S2E [14]. DataGuard adapts LLVM's loop canonicalization feature [53] and S2E's symbolic state merging [70] to reduce the effect by potential path explosion problem. DataGuard limits the scope of symbolic execution by limiting the depth of the call stack, currently set to four functions. This depth was chosen based on angr's CFGEmulated method supporting a depth of three [17], [18] and experience that the depth of four functions is practical. The depth is configurable.

As buffers are often processed using functionality provided by the C library, we also evaluate the uClibc library in the DataGuard safety analyses. uClibc is a lightweight C library for developing embedded Linux systems, and nearly all programs that work with glibc work with uClibc without modifications, even with other shared libraries and multi-threading. KLEE, the symbolic execution engine for S2E, already has a model for uClibc but lacks such a model for glibc. But even with uClibc, KLEE symbolic execution fails for some library functions in some cases. DataGuard maintains a list of unsafe library functions; if unsafe library function is called, we assume related memory object/pointer is unsafe.

VII. Evaluation

~~In this section, we examine the ability of DataGuard to improve the protection of safe stack objects, determine how the steps in the DataGuard approach impact the validation of stack objects, and assess how the application of DataGuard impacts the security and performance of programs. In this evaluation, we examine several server programs, nginx-1.18.0, httpd-2.4.46, proftpd-1.3.7, openvpn-2.5.2, and opensshd-8.6, and the SPEC CPU2006 benchmark suite[5].~~

~~A. Stack Object Safety Comparison~~

~~**Q1**: *How does DataGuard impact the security of safe stack objects compared with prior work?* Table I shows the counts and percentages of safe stack objects found using the NesCheck framework's CCured implementation, Clang Safe Stack, and DataGuard methods.~~

~~As shown in Table I DataGuard classifies 91.45% of stack objects as safe in server programs and the SPEC CPU2006 benchmarks on average. After excluding unsafe~~

[5]We examined 16 out of 19 benchmarks in SPEC CPU INT 2006 and SPEC CPU FP 2006 that are written in C or C++. The remaining benchmarks (xalancbmk, povray and dealII) are not supported by SVF.