CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump

Jun Xu[†], Dongliang Mu[‡†], Ping Chen[†], Xinyu Xing[†], Pei Wang[†], and Peng Liu[†]
[†]The Pennsylvania State University, University Park, PA, USA
[‡]Nanjing University, Nanjing, China
{jxx13,dzm77,pzc10,xxing,pxw172,pliu}@ist.psu.edu

ABSTRACT

After a program has crashed and terminated abnormally, it typically leaves behind a snapshot of its crashing state in the form of a *core dump*. While a core dump carries a large amount of information, which has long been used for software debugging, it barely serves as *informative* debugging aids in locating software faults, particularly memory corruption vulnerabilities. A memory corruption is a special type of software fault that may lead to manipulation of the content at a certain memory. As such, a core dump may contain a certain amount of corrupted data, which increases the difficulty in identifying useful debugging information (*e.g.*, a crash point and stack traces). Without a proper mechanism to deal with this problem, a core dump can be practically useless for software failure diagnosis.

In this work, we develop `CREDAL`, an automatic debugging tool that employs the source code of a crashing program to enhance core dump analysis and turns a core dump to an *informative* aid in tracking down memory corruption vulnerabilities. Specifically, `CREDAL` systematically analyzes a potentially corrupted core dump and identifies the crash point and stack frames. For a core dump carrying corrupted data, it goes beyond the crash point and stack trace. In particular, `CREDAL` further pinpoints the variables holding corrupted data using the source code of the crashing program along with the stack frames. To assist software developers (or security analysts) in tracking down a memory corruption vulnerability, `CREDAL` also performs analysis and highlights the code fragments corresponding to data corruption.

To demonstrate the utility of `CREDAL`, we use it to analyze 80 crashes corresponding to 73 memory corruption vulnerabilities archived in Offensive Security Exploit Database. We show that, `CREDAL` can accurately pinpoint the crash point and (fully or partially) restore a stack trace even though a crashing program stack carries corrupted data. In addition, we demonstrate `CREDAL` can potentially reduce the manual effort of finding the code fragment that is likely to contain memory corruption vulnerabilities.

Keywords

Core Dump; Memory Corruption; Vulnerability Analysis

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978340

1. INTRODUCTION

Despite best efforts of software developers, software inevitably contains defects. After a software defect is triggered, and a program has terminated abnormally, it typically leaves behind a snapshot of its crashing state. In general, the snapshot of a crashing state is organized in the form of a *core dump*, which oftentimes contains the crashing program stack, the final values of local and global variables, and the final values of processor registers.

Since a core dump carries certain clues as to a program crash, commercial software vendors oftentimes utilize it to facilitate failure diagnosis and classify crashes likely caused by the same defect [20, 21, 24]. For example, Microsoft's tool `RETracer` [19] parses a core dump and extracts information such as the crash point and the crashing stack. Then, it employs a backward taint analysis technique to infer program faults and further triages program crashes. While shown to be effective in spotting the function that contributes to a crash, existing technical approaches (*e.g.*, [19, 21]) are less likely to be effective in identifying some program faults, particularly memory corruption vulnerabilities (*e.g.*, buffer overflow and use after free).

A memory corruption vulnerability is a special type of fault in software that could lead to unintentional modification to the content at a memory location and thus compromise the data dependency of a running program. As such, a core dump may carry a certain amount of corrupted data when a memory corruption vulnerability is triggered and incurs a program crash. Since corrupted data can be anywhere in the memory, it leaves a significant challenge for identifying debugging information. In attempting to exploit a buffer overflow vulnerability, for example, an attacker typically overwrites adjacent memory locations. As we will show later in Section 2, this may significantly increase the difficulty in identifying a stack trace and even spotting the crash point. Since a crash point and stack trace are the most useful information for failure diagnosis, without a proper mechanism to locate them in a core dump, a core dump is practically useless.

In fact, existing core dump analysis techniques can barely serve as *informative* debugging aids in locating a memory corruption vulnerability, even though there is no impediment in tracking down the crash point and stack traces of a crashing program. As is mentioned above, a memory corruption vulnerability allows an attacker to compromise the data dependency of a running program. In facilitating failure diagnosis, existing techniques typically perform backward program analysis starting from the crash point, and assume the integrity of a crashing stack is not compromised (*e.g.*, [19, 26]). When such techniques intersect corrupted data, therefore, they may terminate unexpectedly and produce no information other than the crash point and stack traces of a crashing program.

In this work, we develop `CREDAL`, an automatic debugging tool to assist software developers in tracking down software faults, par-

ticularly memory corruption vulnerabilities. Our goal is not to let CREDAL pinpoint a memory corruption vulnerability, but rather to turn a core dump to an *informative* aid in locating the vulnerability. To deal with the challenges that memory corruption introduces to core dump analysis, technically, CREDAL leverages the source code of the crashing program to enhance core dump analysis.

Since the state of the program at the crash is an almost-necessary starting point [30] and stack traces can potentially narrow down the list of candidate files that are likely to contain software defects [38], CREDAL first identifies the crash point and attempts to restore the stack trace of a crashing program.

In general, it is easy to identify a crash point. When a program has crashed and terminated unexpectedly, the final value of the program counter typically indicates the crash point. However, memory corruption may manipulate the program counter, making it point to an invalid instruction. (*e.g.*, overwriting a return address on the stack with a non-executable memory location). To address this problem, CREDAL checks the validity of the program counter. For the invalid program counter, CREDAL restores its value by analyzing the remnants on the stack. More specifically, CREDAL first attempts to identify the function which was just called but silently returned before the crash, in that this function carries the information about its parent (*i.e.*, the crash function). Using this function, CREDAL then locates the crash function as well as the crash point within it. In recovering the crashing stack, CREDAL makes conservative inference using the restored program pointer along with call frame information [18].

As is discussed above, memory corruption can manipulate the content at a certain memory location, which may result in the violation of data dependency. Intuition suggests that highlighting data dependency mismatches seems informative for failure diagnosis. As a result, we further augment CREDAL with the ability of specifying data dependency mismatches at the source code level. In particular, CREDAL identifies the variables – the values of which in memory mismatch the data dependency of the crashing program – and highlights the source code corresponding to the mismatches. Technically speaking, CREDAL first constructs an inter-procedural control flow graph based on the stack traces restored as well as the source code of the crashing program. Then, it performs inter-procedural points-to analysis and reaching definition analysis to discover the mismatches in variable values and pinpoint the code fragments corresponding to the mismatch.

We implemented CREDAL for Linux systems on x86 platform. To the best of our knowledge, CREDAL is the first automatic tool that can perform core dump analysis in the condition where a core dump contains a certain amount of corrupted data. We manually analyzed 80 crashes corresponding to 73 memory corruption vulnerabilities collected from Offensive Security Exploit Database Archive [14], and compared our manual analysis with the analysis conducted by CREDAL. We observed that CREDAL can accurately identify a crash point and (fully or partially) recover stack traces from a core dump. In addition, we demonstrated that CREDAL can potentially increase the utility of a core dump. For about 80% of the crashes, CREDAL can narrow down vulnerability diagnosis within a couple of functions. For about 50% of the crashes, CREDAL can bound diagnosis efforts in only tens of lines of code.

In summary, we make the following contributions.

- We designed CREDAL, an automatic debugging tool that leverages the source code of the crashing program to enhance core dump analysis and provides useful information for software failure diagnosis.
- We implemented CREDAL on Linux for facilitating software developers (or security analysts) to locate software faults, particularly memory corruption vulnerabilities.
- We demonstrated the utility of CREDAL in facilitating memory corruption vulnerability diagnosis by using 80 crashes attributable to 73 memory corruption vulnerabilities.

The rest of the paper is organized as follows. Section 2 defines the problem scope of our research. Section 3 presents the overview of CREDAL. Section 4 and 5 describe the design and implementation of CREDAL in detail. Section 6 demonstrates the utility of CREDAL. Section 7 surveys related work followed by some discussion on CREDAL in Section 8. Finally, we conclude this work in Section 9.

2. PROBLEM SCOPE

In this section, we define the problem scope of our research. We first discuss our threat model. Then, we demonstrate how a memory corruption vulnerability can undermine the utility of a core dump with a real world example.

2.1 Threat Model

Our research focuses on diagnosing the crash of a process. Therefore, we exclude the program crashes that do not incur the unexpected termination of a running process (*e.g.*, Java program crashes). Because our research diagnoses a process crash through core dump analysis, we further exclude the process crashes that typically do not produce core dumps. Up to and including Linux 2.2, the default action for CPU time limit exceeded, for example, is to terminate the process without a core dump [11].

As is mentioned above, our research is motivated by memory corruption. As a result, we only deal with process crashes caused by memory corruption vulnerabilities. Although many software defects can trigger a process crash, and CREDAL can provide useful information for diagnosing any process crashes, the software defects that can trigger a program crash but not result in memory corruption are out of our research scope. In general, such defects include buffer over-read, null pointer accesses, uninitialized variables, and out-of-memory errors. We believe this is a realistic threat model because (1) it covers all the memory corruption vulnerabilities and (2) techniques to analyze excluded software defects have been proposed by other researchers and can be combined with CREDAL.

Note that we design CREDAL as a debugging tool to analyze crashes triggered by memory corruption during random exercises. We do not assume CREDAL can act as a defense meant to work in an adversarial setting where the attackers can actively prevent offline debugging.

2.2 Motivating Example

We use a real world vulnerability – CVE-2013-2028 [2] – as a typical example to illustrate how and why a memory corruption vulnerability can compromise the integrity of a program counter and tamper data on the stack, making a core dump futile for software debugging.

Table 1 shows a code fragment from Nginx-1.4.0. As is described in CVE-2013-2028, this code fragment can manipulate a signed integer and trigger a stack based overflow. More specifically, an attacker can craft a request and thus manipulate the value of r->headers_in.content_length_n. When handling this specifically crafted request, as is shown in line 15, a worker process compares the value held in r->headers_in.content_length_n with a constant, chooses the minimum and assigns it to variable size. Then, the worker process uses this variable to determine the number of bytes it needs to copy from memory area r->connection to memory area buffer (see line 17).

```
1
2  static int ngx_http_read_discarded_request_body
3  (ngx_http_request_t *r){
4     size_t size;
5     ssize_t n;
6     ngx_int_t rc;
7     ngx_buf_t b;
8     u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
9     ...
10    ngx_memzero(&b, sizeof(ngx_buf_t));
11    b.temporary = 1;
12    for ( ;; ) {
13       ...
14       // Choose the minimum value between the
             two arguments
15       size = (size_t) ngx_min(r->headers_in.
             content_length_n,
             NGX_HTTP_DISCARD_BUFFER_SIZE);
16       //copy data to buffer
17       n = r->connection->recv(r->connection,
             buffer, size);
18       ...
19       b.pos = buffer;
20       b.last = buffer + n;
21       rc = ngx_http_discard_request_body_filter
             (r, &b);
22       if (rc != NGX_OK) {
23          return rc;
24       }
25    }
26 }
```

Table 1: A code fragment from Nginx-1.4.0 vulnerable to an integer overflow (CVE-2013-2028).

In this example, `r->headers_in.content_length_n` is a signed integer type, whereas variable `size` is an unsigned type. This inconsistency can potentially triggers a stack overflow which may further results in a crash and even arbitrary code execution. Specifically, an attacker can set `r->headers_in.content_length_n` to negative. Due to the inconsistency in variable type, the value of `r->headers_in.content_length_n` can be misinterpreted as a very large positive number when casting to unsigned type variable `size`. As is defined in `line 8`, array `buffer` can only carry `NGX_HTTP_DISCARD_BUFFER_SIZE` bytes of data. Since the value in `size` is larger than constant `NGX_HTTP_DISCARD_BUFFER_SIZE`, the request can overflow `buffer` and corrupt the data on the stack.

Figure 1 illustrates part of information in a core dump after exploiting the aforementioned vulnerability with a public PoC [13]. We observe the exploit overflows `buffer` and corrupts the local variables, argument and return address of function `ngx_http_read_discarded_request_body`. For this specific example, these corrupted data on the stack do not interrupt the process execution until the function returns. At the time the function returns, the running process simply restores frame pointer `EBP` to the previous value held in it[1], and sets program counter `EIP` to the return address on the stack. As such, we observe both `EBP` and `EIP` are set to an invalid address (`0x41414141`) at the time of the crash.

Since `EBP` is designed to provide a frame pointer for the current function, and `EIP` holds the address of the next CPU instruction,

[1]Note that most compilers provide an option to omit frame pointers. With that option enabled, it makes debugging even more difficult. To illustrate the difficulty in identifying debugging information in a core dump, we assume software developers can retrieve information from `EBP` to assist their failure diagnosis.

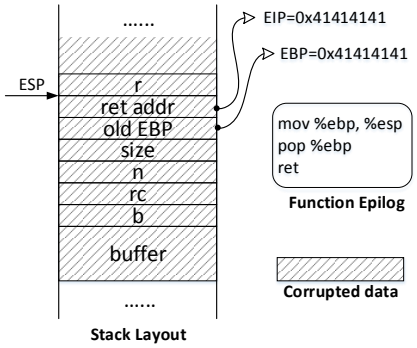


Figure 1: The status of the crashing stack and processor registers after exploiting the overflow vulnerability specified in CVE-2013-2028. For simplicity and demonstration, the stack protector has been disabled.

at the time of a program crash, their snapshots typically indicate the crash function and crash point, respectively. With both process registers holding an invalid address at the time of the crash, however, software developers receive no clue as to the program crash. In the following sections, we will therefore develop new technical approaches to identify the crash point as well as the stack traces of a crashing program.

3. OVERVIEW

In this section, we discuss our design principle followed by the description about how `CREDAL` performs core dump analysis at a high level.

3.1 Design Principle

When locating a software defect, it is always beneficial for software developers to narrow down the manual efforts to code with as few lines as possible. Ideally, we would like to minimize the manual effort of a developer by designing `CREDAL` to pinpoint a software defect directly. However, a core dump may carry a certain amount of corrupted data, and the information held in it only provides a partial chronology of how the program reached a crash point. To track down a software defect, therefore, we need to design `CREDAL` to infer the ambiguity about program execution. This potentially increases the uncertainty in the information that `CREDAL` provides to the developers. Considering such uncertainty may mislead failure diagnosis, our design follows a conservative principle – *maximizing the reliability of the information that* `CREDAL` *produces by minimizing the uncertainty in core dump analysis*.

3.2 Technical Approach

In an extreme case, we can design `CREDAL` to achieve zero uncertainty in core dump analysis by giving no information to software developers. However, such a design sacrifices the utility of a core dump in failure diagnosis. To balance between utility and uncertainty, we therefore design `CREDAL` as follows.

As mentioned in Section 1, a crash point typically serves as the starting point of failure diagnosis, and a stack trace can narrow down the list of candidate files that possibly contain software defects. As a result, we design `CREDAL` to provide this essential information needed by software developers and security analysts. Considering memory corruption discards data dependency, and presenting data dependency mismatch may facilitate failure diagnosis, we also design `CREDAL` to highlight the code fragments corresponding to data dependency mismatch.

To track down the crash point, CREDAL extracts the final value of the program counter stored in a core dump simply because a program counter at the crash reveals where a program crash occurred. As mentioned in Section 2, a memory corruption vulnerability may overwrite a program counter to an invalid value. Before using it to pinpoint the crash point, CREDAL therefore checks the validity of the program counter. For the program counter with an invalid value, CREDAL attempts to restore its value using the data within a previously returned stack frame. We will describe more details in the following section.

To identify a stack trace, CREDAL follows the DWARF standard [18] to unwind a crashing stack. CREDAL can traceback all the functions that have been called but not yet returned at the time of the crash. Note that memory corruption undermines the data on the stack and may thwart stack frame identification. Following the design principle above, CREDAL stops stack frame identification and produces a partial stack trace when identifying a stack frame that does not match certain heuristics. Figure 2 shows one situation where a stack trace cannot be fully identified in an accurate manner and CREDAL terminates in advance. The program crashes in function `crash()`. Using the remnants stored on its stack frame, CREDAL computes the Canonical Frame Address (CFA) and tracebacks to parent function `foo()`. However, a stack overflow occurred in `foo()`, overwrote the data held in the frame and made it invalid. Considering that the program may have an earlier call to `bar1()` or `bar2()`, and there is insufficient information about the execution path in the lead-up to the crash, CREDAL terminates stack trace identification and outputs a partial stack trace with function `crash()` and its caller `foo()`.

To pinpoint data dependency mismatch, CREDAL first constructs a control flow graph (CFG). Given the graph along with the aforementioned crash point and stack trace, CREDAL further performs an inter-procedural points-to analysis and an inter-procedural reaching definition analysis. These analyses allow CREDAL to obtain a set of data dependency constraints and thus identify the variables with mismatching dependency. For the variables with mismatching values, CREDAL further highlights the code fragments corresponding to the mismatch, and presents the code fragment with minimal number of lines of code to software developers (or security analysts). The intuition here is that these code lines may be potentially used as reference for locating a memory corruption vulnerability. In the following section, we will discuss CREDAL with more technical details.

4. DESIGN

In this section, we discuss the technical details of CREDAL. Specifically, we start with crash thread identification. Then, we describe how CREDAL identifies the crash point, stack trace and data dependency mismatch in detail. In addition, we specify the uncertainty that CREDAL may introduce in core dump analysis, and discuss how we leverage technical approaches to minimize this uncertainty.

As is mentioned earlier, a core dump carries the values of processor registers and the values stored in memory, which can be directly consumed by binary-level analysis. As such, we perform core dump analysis mainly on binaries. Considering data flow analysis is typically performed on source code, and source code is self-evident for software developers, we therefore translate the information derived from binary-level analysis into a form that is amenable to source code level analysis. In Section 5, we will describe how to implement CREDAL to map the values in memory and x86 instructions to variables and the statements in source code.

4.1 Discovering Crashing Thread

A process may contain multiple threads. When it crashes, an operating system includes recorded state of the working memory of each thread in a single core dump. To provide useful, interesting information for crash diagnosis, CREDAL first needs to identify the crashing thread in the core dump.

In this work, CREDAL employs the state of the program counter to identify a crashing thread. In particular, CREDAL examines the program counter of each thread at the crash. When CREDAL discovers a program counter that points to an invalid memory address or an illegal instruction (*e.g.*, the instruction containing an invalid opcode or incurring a floating point exception), CREDAL deems the corresponding thread as the one that crashes the process. For the situation where a program counter points to a valid instruction but the instruction attempts to access an invalid memory address, CREDAL also treats the corresponding thread as the one contributing to the crash.

4.2 Tracking down Crash Point

As is mentioned in Section 3, a crash point is typically enclosed in the program counter at the crash. Due to the corrupted data on stack and in processor registers, the program counter may hold an invalid value, making crash point identification difficult.

Here, we deal with this technical issue using the previously returned stack frames. The intuition here is that a crash function generally does not overwrite the stack frame allocated for the earlier function call, and the data on this stack frame can facilitate the identification of the crash point. Figure 2 illustrates an example where a program unexpectedly terminated in function `crash()` but the data on the stack frame corresponding to an earlier call to `bar()` has not yet been overwritten. Since the return address of function `bar()` is stored on its stack frame, and directly points to the next instruction that would be executed in function `crash()`, the crash function can be easily identified through this linkage.

However, a crashing stack does not provide sufficient information that can help pinpoint previously returned stack frames on a crashing stack. To address this problem, we scan a crashing stack through a sliding window, looking for the return address of the function that was just called (but presumably silently returned) before the crash. The intuition here is that a function pushes the return address of its child on the stack at the time the child is invoked, and we use the child as the indicator of the stack frame corresponding to the function.

In this work, we set the size of the sliding window to memory address width (*e.g.*, 4 bytes for a 32-bit operating system). The scan of the crashing stack starts from the top of the stack indicated by the value of stack pointer `ESP` plus an offset equal to the memory address width (*e.g.*, `ESP+4` for a 32-bit operating system).

CREDAL follows two criteria when determining if the value held in the sliding window represents the return address of the previously returned function. As a return address points to the instruction that would be executed after a function returns, CREDAL first ensures the value in the window links to a valid instruction. Second, CREDAL examines the instruction above the one corresponding to the return address. In particular, CREDAL checks if that instruction is a `call` instruction because a `ret` instruction indicates the completion of a function call.

For the value in the sliding window that matches the criteria above, CREDAL deems it as a valid return address candidate, and follows the aforementioned steps to pinpoint the crash function. With the crash function identified, CREDAL further performs program counter recovery. Since memory corruptions manipulate the program counter through indirect jump instructions (*e.g.*, `ret;`

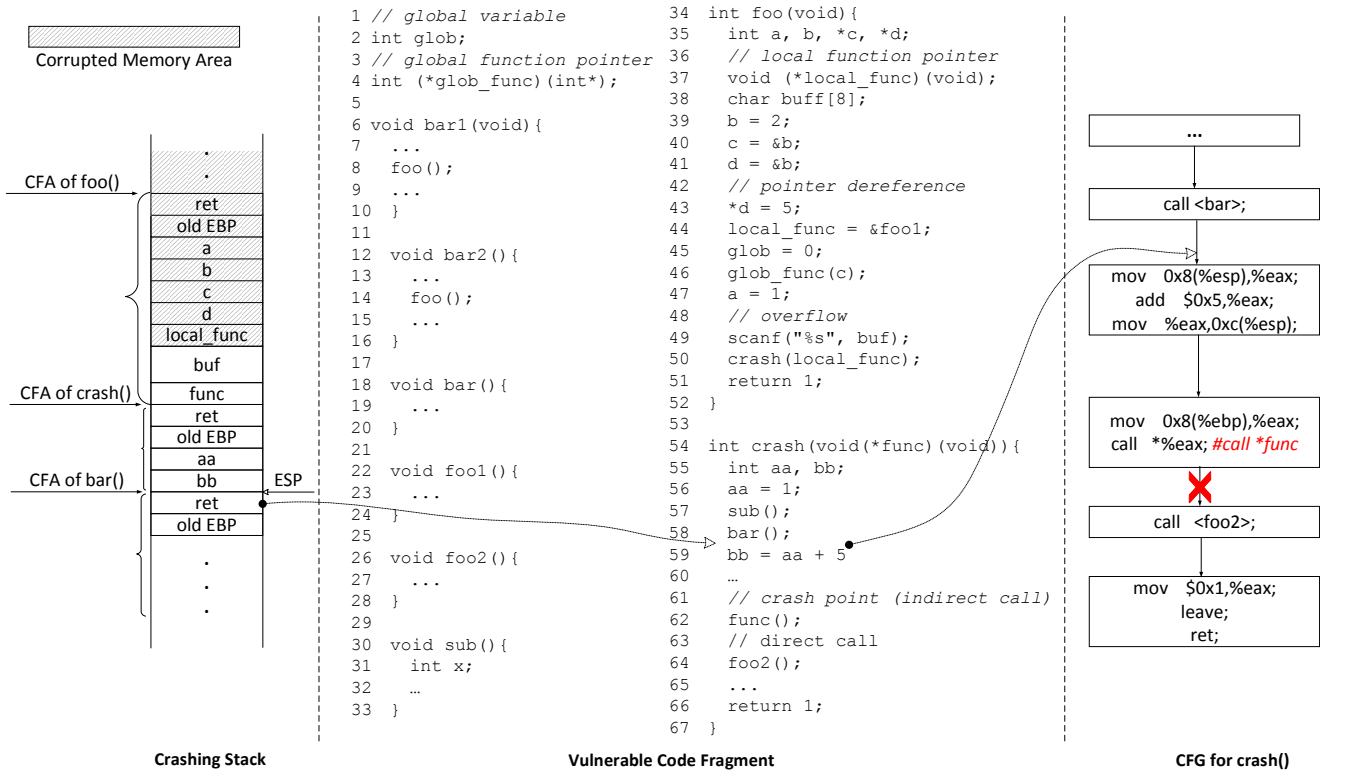


Figure 2: The snapshot of a crashing program after exploiting its overflow vulnerability. For simplicity and demonstration, the stack protector has been disabled and we convert the instruction on a binary into the statement in C programming language.

`call EDX`), and point it to an invalid memory address, CREDAL examines indirect jump instructions in the crash function.

In particular, CREDAL first uses an intra-procedural Control Flow Graph (CFG) to identify all the indirect jump instructions reachable from the instruction corresponding to the candidate return address. Figure 2 shows an intra-procedural CFG for function `crash()`. In this CFG, we prune all the direct function calls and cut off corresponding connections. The intuition here is that the crashing stack exhibits a different layout if the crashing function calls another subroutine after the instruction corresponding to the return address. Again, take the example code shown in Figure 2. The return address on the crashing stack points to an instruction in crash function `crash()`. The CFG within the crash function indicates a call to function `foo2()`. If the program invokes function `foo2()` and crashes after, the stack frame of function `bar()` would not be presented on the crashing stack.

Second, CREDAL verifies the destinations of indirect jump instructions identified on the aforementioned CFG. More specifically, CREDAL computes the destination of each indirect jump instruction using the values of processor registers or memories preserved in the core dump. Then, CREDAL attempts to match the destination with the value held in the program counter. When identifying a match, CREDAL restores the program counter to the address of that indirect jump instruction and deems it as the crash point. Note that if the information is incomplete for CREDAL to recover the crash point, our analysis terminates.

The aforementioned program counter recovery mechanism follows systematic analysis and verification. However, it still introduces uncertainty to crash point identification. For example, a crash function does not invoke any subroutine before the crash, and CREDAL mistakenly identifies the remnants on the stack as a valid

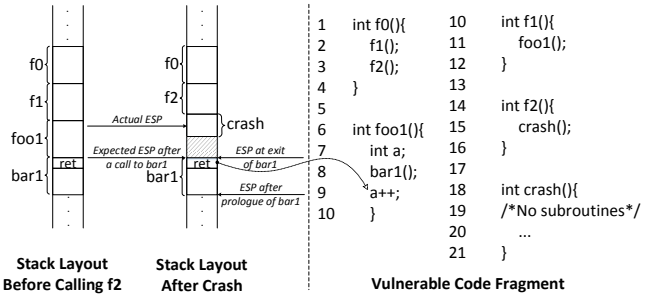


Figure 3: Stack pointer verification. For simplicity and demonstration, the return address points a line of C programming code converted from the instruction on a binary.

return address. To minimize such uncertainty, CREDAL further verifies the identified crash point by checking the displacement of the stack pointer. Figure 3 shows an example where crash function `crash()` does not invoke any subroutines but the stack frame of an earlier call to `bar1()` is preserved. Thus, function `foo1()` is mistakenly identified as the crash function. By examining the displacement of the stack pointer before and after the call to `bar1()`, however, CREDAL can identify the difference between the expected and actual stack pointer. As is illustrated in Figure 3, the position of ESP should be at the bottom and the top of the stack frame of `bar1()` at the entry and exit of the function, respectively. Since there is no other operation to ESP within function `foo1()`, the expected position for ESP should remain at the top of the frame of `bar1()` at the crash. However, this expectation does not match the observation from the core dump, which indicates the incorrectness

of crash point identification. The intuition behind this verification is that the stack frame of the crash function mistakenly identified generally does not share the same size as that of the actual crash function.

With the design discussed above, we believe CREDAL can identify a crash point with high confidence because it is very unlikely to bring about the coincidence where (1) the remnants on the crashing stack are mistakenly deemed as an address that points to a valid instruction; (2) the instruction above the valid instruction is a `call` instruction; (3) an indirect jump in the crash function mistakenly identified encloses a destination that happens to share the same value as the corrupted program counter; (4) the stack frame of the crash function mistakenly identified happens to share the same size as that of the actual crash function. In Section 6, we will demonstrate the effectiveness and correctness of CREDAL in tracking down a crash point.

4.3 Identifying Stack Trace

With the crash point identified, we now discuss how to use it to track down a stack trace. As is described in Section 3, CREDAL has the access to the source code of a crashing program. Thus, it can compile the code with debugging options enabled, and obtain the call frame information of the crashing program.

The call frame information is typically used for stack unwinding. Following DWARF standard [18], therefore, CREDAL can "virtually" unwind a crashing stack and track down a stack trace. As is discussed in Section 3, data corruption on stack however may introduce uncertainty to stack unwinding. As a result, CREDAL also verifies the legitimacy of a stack frame in each step in addition to following the restored registers to find all stack frames.

More specifically, CREDAL walks the crashing stack and checks the validity of the return address in each stack frame by following the criteria discussed in Section 4.2. In addition, CREDAL examines the allocation of a newly unwound stack frame and ensures it is laid just on top of the last stack frame successfully identified. The intuition here is that the frames on stack should be compactly laid out but not overlapped. Similar to the approach we leverage in Section 4.2, CREDAL finally verifies the size of a newly identified stack frame using the displacement of the stack pointer.

As is discussed in the previous section, the design of CREDAL follows a conservative principle. When "virtually" unwinding a crashing stack and identifying a stack frame cannot pass the aforementioned verification, CREDAL stops the unwinding operation and produces a partial stack only with the stack frames successfully identified. We design CREDAL to conservatively identify stack trace so that corrupted stacks can also be handled. In Section 6, we will demonstrate the correctness of CREDAL in partially (or fully) identifying a stack trace.

4.4 Discovering Data Dependency Mismatch

As is mentioned above, memory corruption typically incurs data corruption. If the value of a variable observed in the core dump does not match any reachable definition, a *data dependency mismatch* is found. Here, we describe how CREDAL pinpoints such a dependency mismatch and highlights the corresponding code fragment.

On a high level, we statically analyze the set of possible values for each variable on the recovered trace and match the possible values with the actual value in the core dump. Assuming our analysis on possible value sets is sound, if the value of a variable indicated by the core dump falls out of the corresponding value set, a memory corruption must have occurred.

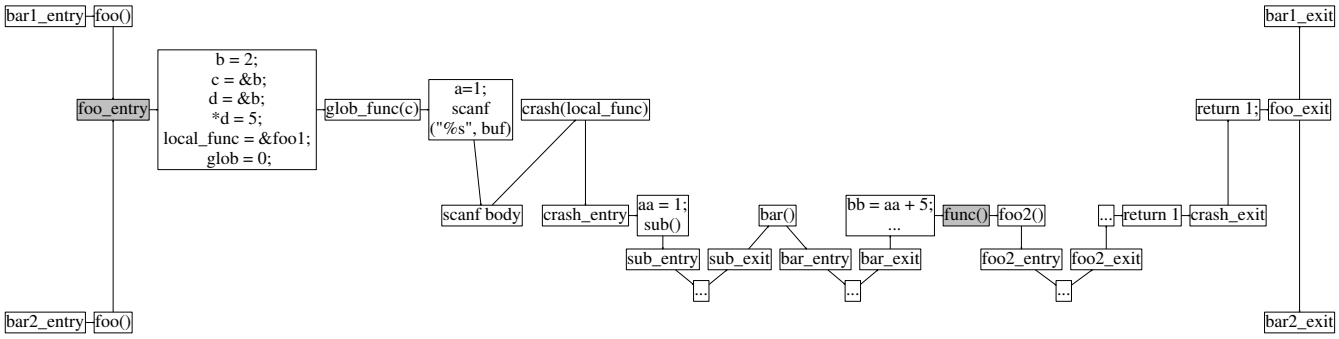
To obtain such value sets, we perform an inter-procedural reaching definition analysis with the restored stack trace. As we will describe in the following presentation, our analysis is conservatively designed to avoid loss of soundness. Specifically, our analysis firstly constructs an inter-procedural CFG that covers all possible call sequences from the entry function to the crash point according to the recovered stack. As resolving indirect calls may introduce inaccuracy, CREDAL skips indirectly called functions (whose targets are unknown) but preserves their arguments. Although skipping indirect calls results in a partial CFG, our subsequent analysis will conservatively consider the potential effects of these indirectly called functions, to make the whole analysis sound. Figure 4 illustrates the CFG corresponding to the example in Figure 2.

To the partial CFG, we apply an intra-procedural points-to analysis to each function, following a context and path insensitive strategy. With the points-to information, we can easily calculate the reaching definition in each function. We then populate the intra-procedural results across function boundaries and extend the results to the whole CFG. We achieve this using a summary-based inter-procedural static analysis algorithm (*i.e.*, the "functional approach") [40]. More specifically, we capture the effects caused by a function modifying variables in another function through pointers passed as arguments. To guarantee the soundness of our results, we handle indirectly called functions in a conservative manner. To be specific, we assume that an indirect call modifies all global variables and all variables possibly pointed to by the argument. We assume those variables may equal any value after the indirect call. Take vulnerable code in Figure 2 for example, our analysis gets rid of global variable `glob` and local variable `b` when analysis reaches to `line 46`, since `glob_func` represents an indirect call which receives pointer `c` to local variable `b`.

After obtaining the reaching definition results, we deduce the possible value set for each variable (*i.e.*, value constraints on each variable). If one definition can be tracked back to a constant value, we add the constant to the set. Otherwise, we assume the definition leads to all possible values. Afterwards we start searching for corrupted variables in the core dump. Note that CREDAL does not consider global variables if the crashing program has multi-threading, because global variables are shared by all threads and a core dump does not unveil when a global variable was modified by other threads. We also do not consider those variables if the crashing stack does not preserve their values (*e.g.*, variable `x` in function `sub()`). The intuition here is that there is no sufficient evidence to examine data dependency mismatch if the final value of the variable is unknown.

For a variable of non-pointer type, if its final value in the core dump does not match the value constraints, we determine a dependency mismatch with this variable. Further, we highlight the code fragment from all the reachable definitions of this variable to the crash point in the CFG. Take variable `a` in Figure 2 for an example. `a` has one definition, namely `a = 1` at `line 47`. The final value of `a` is corrupted and deviates from `1`, thus we catch a mismatch on `a`. The code segment is determined as from `line 47` to `line 50` and `line 55` to `line 62`.

For a variable of pointer type, CREDAL first takes it as a normal non-pointer variable and performs the above check. In addition, CREDAL searches for another type of dependency mismatch. Assuming on any path from the first function to the crash point on the CFG, there exists at least one dereference to this pointer without subsequent re-definition, the pointer must have been unintentionally manipulated. *The intuition is that if there is no unintended manipulation, the process should have crashed in the previous dereferences*. CREDAL deems this as dependency mismatch on pointer dereference. Similarly, CREDAL highlights the code fragment from all the dereferences to the crash point in the CFG.



Constraints: {a@foo = 1; d@foo = [valid]; aa@crash = 1; bb@crash = aa@crash+5}

Figure 4: An inter-procedural control flow graph and data dependency constraints. Note that `scanf()` is a call to an external library. For simplicity and demonstration, we do not unfold this call. The list of constraints defines data dependency *e.g.*, `a@foo=1` indicates variable `a` in function `foo` should be equal to 1 at the crash; `d@foo=[valid]` indicates pointer `d` should be valid at the crash.

5. IMPLEMENTATION

We have implemented a prototype of CREDAL for Linux 32-bit system, which takes as input a core dump file as well as the binary and source code of the corresponding crashing program. As CREDAL needs debugging information for analysis, our implementation requires the binary to be compiled with debugging options (*e.g.*, `-g` with `gcc`). In this section, we present some important implementation details.

Linux operating system organizes a core dump file in the form of Executable Linkable Format (ELF). The implementation of CREDAL employs `libelf`, an open source library [9] to parse the file in ELF and retrieve the corresponding memory information. Considering CREDAL needs to examine the entire working memory of a crashing program, and Linux kernel typically does not include file-backed mappings in a core dump, our implementation augments `libelf` with the ability to interpret the note segment in an ELF file so that CREDAL can identify file-backed memory mappings and consume the information in that memory area.

CREDAL currently relies on the debug information to disassemble binaries and unwind crashing stacks. For disassembly, our implementation uses `libdwarf` library [6] to parse a binary and then employs `libdisasm` library [8] to identify instructions in it. For unwinding a crashing stack, our implementation relies on `libdwarf` library to extract call frame information from `.debug_frame` and `.eh_frame` stored in ELF files. To perform virtual unwinding, we also implement CREDAL by modifying `libunwind` library [10].

As is mentioned in Section 4, CREDAL constructs an intra-procedural CFG on a binary. However, indirect jump instructions introduce non-deterministic to the CFG construction. Given instruction `[jmp %EAX]`, for example, it is difficult to construct the consecutive nodes on CFG without determining the destination of this instruction. To address this problem, our implementation uses LLVM [12] to extract program semantics from source code and identify the destinations of indirect jump instructions. For example, assume an indirect jump instruction is a low-level representation of a `switch` statement in C programming language. Our implementation employs LLVM APIs to identify the destinations from those `case` statements, and completes CFG construction.

In our design, CREDAL utilizes the displacement of a stack pointer to verify the crash point and stack trace identified. To do this, CREDAL needs to know the change to `ESP` for a given code fragment. Within the code fragment, there may be a variety of execution paths that cover the operations of `ESP` (*e.g.*, `[add $0x4, %ESP]`, `[pop]` or `[push %EAX]`). Theoretically, stack pointer `ESP` may end up at different position when going through different paths. To guarantee the correctness of program execution, a compiler however ensures `ESP` has the same displacement whichever paths the program walks through. As a result, our implementation chooses an arbitrary path to compute the displacement of `ESP` when verifying a crash point or stack trace.

Our implementation seeks for data dependency mismatches on LLVM IR. We construct the call graph in an on-demand manner. We compile each source code file separately into LLVM IR and traverse from the first function on the recovered stack trace to the crash point. Whenever a function is directly called, we include that function and expand the call graph. If that function is in a different IR file, we search for it in other compile units by its name and linkage type. With the call graph and the intra-procedure CFG natively provided by LLVM, we essentially have the inter-procedure CFG.

To the CFG, we apply LLVM's built-in basic alias analysis to get intra-procedure point-to results. To ensure the soundness of the analysis, our implementation takes `may-alias` relation as `must-alias` relation. Though the conservativeness theoretically limits the analysis capability of CREDAL, our case studies in Section 6 demonstrate that this implementation can work well in practice. LLVM also provides intra-procedural reaching definition results through its `use-def` chain data structures. With these two pieces of information, we further implemented a simple summary-based inter-procedural analysis as described in Section 4.

As is discussed in Section 4, CREDAL examines variables in LLVM IR, the values of which are preserved in the core dump. Our implementation utilizes debugging information in the IR to achieve mapping between source code variables and LLVM IR variables and we again leverages debugging information in the binary to compute the locations of source code variables in the core dump. We highlight the code segment for data dependency mismatch in LLVM IR, which is also mapped to source code segment via debugging information in the IR file. Source code of CREDAL is available at https://github.com/junxzm1990/credal.git.

6. CASE STUDY

In this section, we demonstrate the utility of CREDAL using the crashes attributable to memory corruption vulnerabilities. In particular, we describe the collection of crashes and present the effectiveness of CREDAL. We also discuss those memory corruption vulnerabilities, the crashes of which CREDAL fails to handle.

Program	Program Size (LOC)	CVE-ID	Vulnerability Type	EDB-ID	EIP	# of Frame	Full Stack	Area (LOC)	Root Cause	# of Function
tack 1.0.7	20979	NA	BSS/Data Overflow	38685	√	2	√	27	√	1
picpuz 2.1.1	6345	NA	BSS/Data Overflow	10634	√	2	√	25	√	2
2Fax 3	2664	NA	BSS/Data Overflow	24984	√	4	√	NA	NA	NA
ytree 1.94	14010	NA	BSS/Data Overflow	39406	√	5	√	NA	NA	NA
nullhttpd 0.5.0	1849	NA	Heap Overflow	218181	√	6	X	15	√	1
make 3.8.1	24151	NA	Heap Overflow	34164	√	1	X	12	√	1
xrdp 0.4.1	33995	2008-5904	Heap Overflow	8469	√	14	X	224	√	3
libsndfile 1.0.25	51064	2015-7805	Heap Overflow	38447	√	26	√	280	√	3
ntpd 4.2.6	152433	NA	Heap Overflow	39445	X	4	√	33	X	7
php 5.3.6	805640	2012-2386	Heap Overflow	17201	√	10	√	NA	NA	NA
inetutils 1.8	98941	NA	Heap Overflow	15705	√	3	√	NA	NA	NA
nginx 1.4.0	100255	2013-2028	Integer Overflow	25775	√	1	X	36	X	3
0verkill 0.16	16361	2006-2971	Integer Overflow	1894	√	5	√	4	X	1
php 5.16	561820	2007-1001	Integer Overflow	29823	√	14	√	46	√	3
ImageMagick 6.2.0	214800	2006-4144	Internet Overflow	28283	√	5	√	110	√	2
php 4.4.4	367295	2007-1777	Integer Overflow	29788	√	6	√	NA	NA	NA
python 2.2	416060	2007-4965	Integer Overflow	30592	√	11	√	NA	NA	NA
ClamAV 0.88.2	42160	2006-4182	Integer Overflow	2587	√	1	X	NA	NA	NA
mcrypt 2.5.8	10363	2012-4409	Stack Overflow	22938	√	1	X	2	√	1
putty 0.66	90165	2016-2563	Stack Overflow	39551	√	1	X	3	√	1
php 5.3.4	803486	2011-1938	Stack Overflow	17318	√	1	X	43	√	1
rsync 2.5.7	19487	2004-2093	Stack Overflow	152	X	1	X	27	√	1
corehttp 0.5.3a	935	2007-4060	Stack Overflow	4243	√	2	X	6	√	1
aireplay-ng 1.2-beta3	62656	2014-8322	Stack Overflow	35018	X	1	X	124	√	1
No-IP DUC 2.19-1	2578	NA	Stack Overflow	25411	X	1	X	87	√	1
opendchub 0.8.1	11021	2010-1147	Stack Overflow	11986	X	1	X	22	√	1
unrar 3.9.3	17575	NA	Stack Overflow	17611	X	1	X	2	X	2
peercast 0.1214	17193	2006-1148	Stack Overflow	1574	X	1	X	52	X	13
peercast 0.1214	17193	2006-1148	Stack Overflow	1578	X	1	X	52	X	13
peercast 0.1214	17193	2006-1148	Stack Overflow	16855	X	1	X	52	X	13
peercast 0.1214	17193	2007-6454	Stack Overflow	30894	√	9	√	261	√	6
ettercap 0.7.5.1	53175	2013-0722	Stack Overflow	23945	√	2	X	32	√	1
prozilla 1.3.6	13070	2004-1120	Stack Overflow	652	√	2	X	17	√	2
prozilla 1.3.6	13070	2004-1120	Stack Overflow	1238	X	1	X	35	√	1
sftp 1.1.0	1559	NA	Stack Overflow	9264	√	6	√	102	X	27
fbzx 2.5.0	15341	NA	Stack Overflow	38681	√	5	√	144	√	5
Conquest 8.2	105784	2007-1371	Stack Overflow	29717	√	1	X	54	√	1
tiffsplit 3.8.2	47769	2006-2656	Stack Overflow	1831	√	2	√	21	√	2
alsaplayer 0.99.76	26834	2007-5301	Stack Overflow	5424	√	3	√	13	X	6
xmp 2.5.1	45404	2007-6731	Stack Overflow	30942	X	1	X	4	√	1
proftpd 1.3.0a	111839	2006-6563	Stack Overflow	3730	X	1	X	79	√	1
proftpd 1.3.0a	111839	2006-6563	Stack Overflow	2928	X	1	X	79	√	1
proftpd 1.3.0a	111839	2006-6563	Stack Overflow	3330	X	1	X	79	√	1
proftpd 1.3.0a	111839	2006-6563	Stack Overflow	3333	X	1	X	79	√	1
vfu 4.1	18734	NA	Stack Overflow	35450	√	3	X	85	√	1
LibSMI 0.4.8	80461	2010-2891	Stack Overflow	15293	√	3	X	23	√	2
FENIX 0.92	25236	NA	Stack Overflow	37987	√	3	√	1	√	1
gif2png 2.5.2	1331	2009-5018	Stack Overflow	34356	√	1	√	7	√	1
hexchat 2.10.0	68181	2016-2233	Stack Overflow	39657	√	1	X	80	√	1
Binutils 2.15	697354	2006-2362	Stack Overflow	27856	√	4	X	44	√	3
alsaplayer 0.99.76	26834	2007-5301	Stack Overflow	5424	√	3	X	13	X	6
glibc 2.12.90	843348	2015-7547	Stack Overflow	39454	√	4	X	230	X	4
gas 2.12	595504	2005-4807	Stack Overflow	28397	√	1	X	4	√	1
nasm 0.98.38	33553	2004-1287	Stack Overflow	25005	X	1	X	12	√	1
ringtonetools 2.22	6507	2004-1292	Stack Overflow	25015	√	2	X	6	√	1
abc2mtex 1.6.1	4052	NA	Stack Overflow	25018	√	1	X	7	√	1
JPegToAvi 1.5	580	NA	Stack Overflow	24981	√	3	X	17	√	3

Program	Program Size (LOC)	CVE-ID	Vulnerability Type	EDB-ID	EIP	# of Frame	Full Stack	Area (LOC)	Root Cause	# of Function
O3read 0.03	932	2004-1288	Stack Overflow	25010	√	5	√	7	X	2
LateX2tf 1.9.15	14473	2004-2167	Stack Overflow	24622	√	2	X	1	√	1
libpng 1.2.5	33681	2004-0597	Stack Overflow	389	√	1	X	1	X	2
unrtf 0.19.3	5039	NA	Stack Overflow	25030	√	7	√	7	√	1
Sox 12.17.4	25736	2004-0557	Stack Overflow	369	√	4	√	83	√	1
Sox 12.17.4	25736	2004-0557	Stack Overflow	374	√	1	X	1	X	2
psutils-p17	1736	NA	Stack Overflow	890	√	2	X	1	√	2
streamripper 1.61.25	27304	2006-3124	Stack Overflow	2274	√	3	X	7	X	3
Newspost 2.1	4865	2005-0101	Stack Overflow	25077	√	3	√	7	X	16
Unalz 0.52	8546	2005-3862	Stack Overflow	26601	√	1	X	1	√	1
proftpd 1.3.0a	72233	2006-5816	Stack Overflow	2856	√	1	X	36	√	1
proftpd 1.3.3a	111839	2010-4221	Stack Overflow	16878	√	1	X	NA	NA	NA
ht-editor 2.0.18	119236	NA	Stack Overflow	17083	X	1	X	NA	NA	NA
0verkill 0.16	16361	2004-0238	Stack Overflow	23634	X	1	X	NA	NA	NA
ht-editor 2.0.20	119688	NA	Stack Overflow	22683	X	1	X	NA	NA	NA
coreutils 8.4	138135	2013-0221	Stack Overflow	38232	√	5	X	NA	NA	NA
vfu 4.1	18734	NA	Stack Overflow	36229	X	3	X	NA	NA	NA
mutt 1.4.2.2	61913	2007-2683	Stack Overflow	30093	√	1	X	NA	NA	NA
fontforge 20100501	551083	2010-4259	Stack Overflow	15732	√	1	X	NA	NA	NA
compface 1.5.2	1574	2009-2286	Stack Overflow	8982	X	0	X	NA	NA	NA
openlitespeed 1.3.19	97241	NA	Use After Free	37051	√	15	√	26	√	1
lighttpd 1.4.15	38102	2007-3947	Use After Free	30322	√	6	√	456	X	28
python 2.7	100975	2009-2286	Use After Free	100975	√	16	√	NA	NA	NA

Table 1: The list of the program crashes corresponding to memory corruption vulnerabilities. *CVE-ID* and *EDB-ID* specify the IDs of the CVE and corresponding PoC, respectively. *EIP* indicates the validity of the program counter at the crash. *# of Frame* and *# of Functions* describe the number of stack frames `CREDAL` identifies as well as the number of functions one needs to examine when locating the corresponding vulnerability. The numbers in *Area (LOC)* indicate the lines of code corresponding to data dependency mismatch.

6.1 Setup

To demonstrate the utility of `CREDAL`, we must collect program crashes contributed by memory corruption vulnerabilities. We exhaustively searched memory corruption vulnerabilities on Offensive Security Exploit Database Archive [14]. Our goal is to gather the crashes by exploiting memory corruption vulnerabilities with corresponding PoCs.

As an outdated vulnerability typically associates with an obsolete program, and such a program may be no longer available, we only gathered memory corruption vulnerabilities archived over the past twelve years. Because we implement `CREDAL` for Linux operating system, we further narrowed down our searches on the vulnerabilities identified on software running on Linux. In total, we obtained 392 memory corruption vulnerabilities bundled with at least one PoC. We compiled and configured vulnerable programs based on the description of the collected vulnerabilities, and successfully produce 80 crashes using the PoCs corresponding to 73 vulnerabilities. The experiments are conducted on a machine with Intel Xeon E5-2560 2.30GHz and 2GB memory running Ubuntu 14.04. The average time to analyze a core dump is `0.21` seconds.

Table 1 lists the aforementioned crashes and summarizes the corresponding vulnerabilities across 5 different categories, including use-after-free as well as overflow on stack, heap, integer and bss/data. These vulnerabilities are identified on 62 distinct software, ranging from sophisticated programs like `PHP` and `Binutils` with lines of code over `670K` to lightweight programs such as `o3read` and `corehttp` with lines of code less than one thousand.

Note that we discard a large fraction of vulnerabilities for three reasons. First, the program corresponding to a vulnerability is obsolete and we cannot find its source code for showcasing the utility of `CREDAL` (*e.g.*, `Gaim` [7] and `Abc2midi` [1]). Second, compiling a vulnerable program requires an obsolete external library that we cannot discover (*e.g.*, `Asterisk` [4] and `Blender` [5]). Third, a vulnerable program is close-source (*e.g.*, `Apple Quicktime` [3], `Sun Java Runtime Environment` [16] and `Safari` [15]).

6.2 Results

To demonstrate the utility of `CREDAL`, we manually analyze the crashes shown in Table 1, and compare our manual analysis with that of `CREDAL`. More specifically, we evaluate the utility of `CREDAL` as follow. First, we verify if `CREDAL` can restore a program counter and correctly identify the crash point when that is overwritten and set to an invalid value. Second, we examine if `CREDAL` can unwind a crashing stack and pinpoint a full (or partial) stack trace in an accurate manner. Last but not least, we investigate how effective `CREDAL` can enclose a memory corruption vulnerability within the functions and code fragment that it highlights.

6.2.1 *Pinpointing Crash Point & Stack Trace*

Table 1 specifies the validity of a program counter at the time of the crash. We observe 21 crashes for which the core dumps carry a program counter with an invalid value. Among these crashes, `CREDAL` is able to restore program counters for 20 crashes, and the program counters recovered all point to crash points correctly.

Table 1 also indicates the quantity of the stack frames that CREDAL identifies. We discovered that CREDAL can fully (or partially) unwind a crashing stack when a crash point is successfully pinpointed. The reason is that the crash point reveals the crash function in a binary, and even in the worst case, CREDAL can leverage debugging information to identify the stack frame of the crash function.

We examined the crash for which CREDAL fails to restore the program counter, particularly the crash of vulnerable program `compface 1.5.2`. In the actual crash function, we observed that the function employs `setjmp()` to save its calling environment before transferring its execution to a subroutine. The subroutine contains a stack overflow vulnerability. When exploited, it overflows the current stack frame as well as those at the higher memory address. The overflow does not block the program execution immediately. Instead, the subroutine invokes `longjmp()` which transfers its execution to a predetermined location in the crash function. In this case, the instruction at the predetermined location causes an unexpected crash because of the data corruption on the stack.

Performing analysis for this crash, CREDAL can only discover the stack frame of `longjmp()` from the remnants in the core dump. Recall that CREDAL identifies a crash point using the function that was just called but silently returned before the crash. In this case, the execution of the crash function is not returned from its child function but a descendant function – `longjmp()`. As such, the displacement verification of a stack pointer fails and CREDAL conservatively produces no information about the crash point.

6.2.2 *Locating Vulnerability*

In Table 1, we also show the lines of code that CREDAL highlights corresponding to data dependency mismatch. For 63 crashes, CREDAL successfully identifies dependency mismatch in memory. Among these crashes, we observe 47 crashes for which CREDAL can enclose the memory corruption vulnerability (*i.e.*, root cause) within the code fragment highlighted. As is shown in Table 1, a code fragment highlighted typically covers the statements with only tens of lines (in about 90% cases). This indicates CREDAL has a high potential to reduce manual efforts for locating a memory corruption vulnerability in a crash.

Within the batch of the crashes shown in Table 1, there are 16 crashes for which CREDAL identifies dependency mismatch but not encloses the root cause within the code fragment highlighted. For these crashes, we manually examined the code fragment highlighted and the function calls it encloses by imitating the way a security analyst locates a vulnerability. In particular, we started from the code fragment and traversed the enclosed calls in a breadth-first manner. Except for integer overflow in `nginx 1.4.0` and `overkill 0.1.6`, we successfully identified all vulnerabilities in the enclosed function calls. Table 1 specifies the number of functions that we walked through when locating a vulnerability. We observe the numbers of the functions we looked into are relatively small, with an average of 3.46. Again, this indicates CREDAL is potentially effective in locating memory corruption vulnerabilities.

In general, overflowing an integer variable does not directly corrupt data in the function where it is enclosed. Rather, it indirectly incurs a buffer overflow and data corruption in a descendant function. For the aforementioned integer overflow, we therefore discovered the overflow vulnerabilities lie outside the code fragment that CREDAL highlights. However, this does not mean CREDAL is less effective in helping security analysts locate integer overflow vulnerabilities. Considering CREDAL typically encloses overflowed buffers in the code fragment, we therefore believe a security analyst can quickly track down integer overflow using the linkage between the overflowed integer variable and the overflowed buffer.

Last but not least, we also manually examined the remaining crashes for which CREDAL fails to identify data dependency mismatch. Except for the one that CREDAL fails to restore the program counter, Table 1 specifies 17 crashes in this category. For 9 of them, the failure of CREDAL results from the conservative design of identifying data dependency mismatch.

For 6 of the crashes, the failure of CREDAL can be attributed to one of the following. First, data corruption occurs in the stack area that CREDAL cannot unwind (*e.g.*, `ht-editor 2.0.18 & 2.0.20`). Second, the corrupted data was sited in local variables that were overwritten by variable assignment operations in consecutive execution (*e.g.*, `proftpd 1.3.3 a` and `vfu 4.1`). Third, data corruption occurs at a certain memory area in which the corrupted data has not yet been initialized before the crash (*e.g.*, `2Fax 3` and `ytree 1.94`).

For the remaining 2 crashes that CREDAL fails to find dependency mismatch, the overflow corresponding to the crashes represents two special cases. In one case, a PoC exploits vulnerable program `mutt 1.4.2.2` and underflows the data on stack. At the crash, the program counter points to a call to `memmove()`. As CREDAL lacks sufficient information to unwind the stack at the higher frame level, and function `memmove()` does not carry local variables, CREDAL produces no dependency constraint and thus provides less information for locating the overflow vulnerability. In another case corresponding to `clamv 0.88.2`, CREDAL fails to identify data corruption simply because the crash occurs in advance of data corruption. More specifically, the exploit attempts to overflow a buffer by copying a large data chunk from an invalid memory address.

7. RELATED WORK

This research work mainly focuses on analyzing program crashes. Thus, the line of work most closely related to our own is crash analysis, in which program instrumentation, program analysis and core dump forensics are typically used to track down a particular fault resided in a program. In this section, we summarize previous studies and discuss their limitation in turn.

Program instrumentation To spot program faults, a large amount of research focuses on failure reproduction using execution traces (*e.g.*, [23, 31, 32, 35, 44, 45]). Technically speaking, the typical approach along this line is to instrument a program, so that it can automatically generate execution traces at run time. By analyzing these execution traces, one can derive control and data flow and thus identify the faults in software. Since this run-time recording scheme provides more information about program execution, it is effective in locating program faults.

Many other works instead instrument programs to spot memory corruptions at run-time, such as AddressSanitizer [39], SoftBound [29], and Code-Pointer Integrity [25]. AddressSanitizer uses shadow memory to record whether a memory access is safe, and relies on instrumentation to verify the shadow memory in `load` or `store`; SoftBound inserts run-time bounds checks to enforce spatial safety using customized disjoint memory metadata; Code-Pointer Integrity would detect when a code pointer is overwritten and terminate the execution. With Code-Pointer Integrity, a core dump will be generated before any illegal control flow transfer and thus, involves less uncertainty for analysis by CREDAL. These techniques aim at detecting corruptions instead of pinpointing the vulnerability areas.

In practice, many of these approaches introduce high overhead during normal operation, which greatly affects their deployment. Considering practicality, our work does not instrument programs, nor rely upon the availability of existing program logging or exe-

cution traces. Rather, our technical approach facilitates program failure diagnosis by using more generic information, *i.e.* the core dump that operating system automatically captures every time a process has crashed or otherwise terminated abnormally.

Program analysis Over the past decades, there is a rich collection of literature on using program analysis techniques along with crash reports to identify faults in software (*e.g.*, [17, 22, 27, 28, 34, 36, 41, 46]). These existing techniques are designed to identify some specific software defects. In adversarial settings, an attacker exploits a variety of software defects and thus they cannot be used to analyze a program crash caused by a security defect such as buffer overflow or unsafe dangling pointer. For example, Manevich *et al.* [27] proposed to use static backward analysis to reconstruct execution traces from a crash point and thus spot software defects, particularly typestate errors [42]. Similarly, Strom and Yellin [41] defined a partially path-sensitive backward dataflow analysis for checking typestate properties, specifically uninitialized variables. While demonstrated to be effective, these two studies only focus on specific typestate problems.

Liblit *et al.*also proposed a backward analysis technique for crash analysis [26]. In particular, they introduce an efficient algorithm that takes as input a crash point as well as a global control flow graph, and computes all the possible execution paths that lead to the crash point. In addition, they discussed how to narrow down the set of possible execution paths using a wide variety of post-crash artifacts, such as stack or event traces. While our work also uses stack traces for crash analysis, our approach is fundamentally different. As is mentioned earlier, memory information might be corrupted when attackers exploit a program. Thus, the path analysis based on stack traces described in [26] fails because its effectiveness highly relies upon the stack integrity. In contrast with [26], our approach leverages the source code of a crashing program to enhance core dump analysis and pinpoints the code statements where a software defect is likely to reside.

Core dump forensics Considering the low cost of capturing core dumps, prior studies [19, 21, 33, 37, 43] proposed to use core dumps to analyze the root cause of software failures. To facilitate software failure debugging, Polishchuk *et al.* [33] for example proposed a mechanism to reconstruct variable types from heap memory, and Salkeld and Kiczales [37] introduced a method to resurrect Java objects from a shadow heap dump. As part of our work, we also restore memory information, but go beyond objects in memory at the time of the program crash.

Stepping over memory semantic reconstruction, Microsoft developed Windows Error Reporting (WER) service [21], which uses a tool – `!analyze` – to examine a core dump and determine which thread context and stack frame most likely caused the error. Although sharing the same goal as our work, `!analyze` cannot handle program crashes caused by security defects for the reason that the attacks may introduce memory corruption and processor register failures and the effectiveness of `!analyze` highly relies on these information.

In recent research, Wu *et al.* [43] proposed `CrashLocator`, a method to locate software defects by analyzing stack information in a core dump. In addition, Cui *et al.* [19] introduced `RETracer`, a system that reconstructs program semantics from core dumps and examines how program faults contribute to program crashes. More specifically, `RETracer` leverages a core dump along with a backward analysis mechanism to recover program execution status and thus spot a software defect. Since the effectiveness of both techniques highly relies upon the integrity of a core dump, and exploiting vulnerabilities like buffer overflow and dangling pointers corrupts memory information, `CrashLocator` and `RETracer` fail to perform crash analysis.

Different from aforementioned core dump forensics, our approach can deal with both corrupted and uncorrupted core dumps and facilitate program failure diagnosis. To the best of our knowledge, our work is the first research that analyzes the core dump of a crashed program without the assumption of memory integrity.

8. DISCUSSION

In this section, we discuss the limitations of our current design, insights we learned and possible future directions.

Other crashes. `CREDAL` is designed for providing useful information for software developers (or security analysts) to diagnose the crashes caused by memory corruption vulnerabilities. However, it is not limited to analyzing the crashes that contain data corruption. For the crashes without data corruption, `CREDAL` only pinpoints the crash point and full stack trace of the crashing program. While such information may not help developers narrow down their debugging efforts within a couple of lines of code, `CREDAL` still improves the utility of a core dump, especially considering the situation where the program counter points to an invalid address and existing techniques fail to recover it.

Multiple threads. `CREDAL` only focuses on analyzing the data of the crash thread and providing information for debugging. However, a program crash may be contributed by multiple threads. Thus, the information from the crash thread may not help software developers downsize the code space that they have to manually analyze. While this multi-thread issue indeed limits the capability of a security analyst utilizing `CREDAL` to track down a security vulnerability, this does not significantly downgrade the utility of `CREDAL`. In fact, a prior study [38] has already indicated that a large fraction of software defects involves only the crash thread. This finding is consistent with our observation from the Offensive Security Exploit Database archive. Looking into the aforementioned vulnerabilities over the past twelve years, we do not discover any vulnerability, the crash of which needs multi-thread coordination.

Potential attacks. When demonstrating the utility of `CREDAL`, we conducted an exhaustive search to find all the PoCs we can experiment with. These PoCs are, however, unaware of `CREDAL`. Real-world attackers who know about `CREDAL` might actively prevent our analysis. For instance, they may thwart crash point recovery and stack trace recovery via erasing the whole stack, and they may also carefully set up memories to avoid data dependency mismatch. We will take it as our future work to study the possibility of counteracting offline debugging.

9. CONCLUSION

In this paper, we develop a debugging tool `CREDAL` to facilitate core dump analysis. With the support from source code, we show that `CREDAL` can enhance core dump analysis and make a core dump more informative for diagnosing software defects, particularly locating memory corruption vulnerabilities. The design of `CREDAL` follows a conservative principle. Thus, it preserves the utility of a core dump, and at the same time, minimizes the uncertainty in core dump analysis.

We demonstrated the utility of `CREDAL` using the crashes corresponding to 73 memory corruption vulnerabilities. We showed that `CREDAL` can accurately pinpoint a crash point as well as a stack trace. In addition, we demonstrated a memory corruption vulnerability typically lies in the code fragment relevant to data corruption. Following this finding, we safely conclude `CREDAL`

can significantly downsize the code space that a software developer (or security analyst) needs to manually examine, especially when memory corruption occurs.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback and our shepherd, Mathias Payer, for his valuable comments on revision of this paper. We also would like to thank Professor Bing Mao for his technical advise. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, NSF CNS-1505664, ARO W911NF-15-1-0576, and NIETP CAE Cybersecurity Grant.

References

[1] Abc2midi 2004-12-04 - multiple stack buffer overflow vulnerabilities. https://www.exploit-db.com/exploits/25019/.

[2] Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (cve-2013-2028). http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[3] Apple quicktime (rtsp url handler) stack buffer overflow exploit. https://www.exploit-db.com/exploits/3064/.

[4] Asterisk <= 1.0.12 / 1.2.12.1 (chan_skinny) remote heap overflow (poc). https://www.exploit-db.com/exploits/2597/.

[5] Blender blenloader 2.x file processing integer overflow vulnerability. https://www.exploit-db.com/exploits/26915/.

[6] Da's dwarf page. https://www.prevanders.net/dwarf.html.

[7] Gaim <= 1.2.1 url handling remote stack overflow exploit. https://www.exploit-db.com/exploits/999/.

[8] libdisasm: x86 disassembler library. http://bastard.sourceforge.net/libdisasm.html.

[9] Libelf - free software directory. https://directory.fsf.org/wiki/Libelf.

[10] The libunwind project. http://www.nongnu.org/libunwind/.

[11] Linux programmer's manual. http://man7.org/linux/man-pages/man7/signal.7.html.

[12] The llvm compiler infrastructure. http://llvm.org/.

[13] Nginx 1.3.9-1.4.0 - dos poc. http://seclists.org/fulldisclosure/2013/Jul/att-90/ngxunlock_pl.bin.

[14] Offensive security exploit database archive. https://www.exploit-db.com/.

[15] Safari 5.02 - stack overflow denial of service. https://www.exploit-db.com/exploits/15558/.

[16] Sun java runtime environment 1.6 - web start jnlp file stack buffer overflow vulnerability. https://www.exploit-db.com/exploits/30284/.

[17] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[18] D. D. I. F. Committee. Dwarf debugging information format (version 4). http://www.dwarfstd.org/doc/DWARF4.pdf, 2010.

[19] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[20] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[21] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[22] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[23] S. Horwitz, B. Liblit, and M. Polishchuk. Better debugging via output tracing and callstack-sensitive slicing. *IEEE Transaction Software Engineering*, 2010.

[24] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, 2011.

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, Oct. 2014. USENIX Association.

[26] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical report, 2002.

[27] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004.

[28] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[30] P. Ohmann. Making your crashes work for you (doctoral symposium). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[31] P. Ohmann and B. Liblit. Cores, debugging, and coverage. Technical report, 2015.

[32] P. Ohmann and B. Liblit. Csiclipse: Presenting crash analysis data to developers. In *Proceedings of the on Eclipse Technology eXchange*, 2015.

[33] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[34] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2003.

[35] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference*, 1997.

[36] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[37] R. Salkeld and G. Kiczales. Interacting with dead objects. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[38] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-sanitizer: a fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[40] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[41] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transaction Software Engineering*, 1993.

[42] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transaction Software Engineering*, 1986.

[43] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[44] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[45] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[46] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.