State Repair From Cyber Attacks

He Zhu
Galois, Inc.

1 Introduction

This paper presents a *Symbolic Execution* and *Abstract Interpretation* based static analysis tool to detect which part of a system state is compromised by a cyber attack and perform a full recovery by leveraging special binary layout information enforced by LLVM-assisted compile time randomization.

We operate within the context of a Multi-Variant Execution Environment (MVEE), which executes multiple versions ("variants") of an application simultaneously. The variants are built to have identical behavior under normal execution conditions, but to behave differently when under attack. For example, variants that each have different data layouts may behave differently when subject to low-level memory corruption attacks. The MVEE monitor can detect these variations in behavior, and invoke our State Repair technology to recover from these attacks. Our state repair approach is designed to recover from various memory corruption attacks (rather than, for example, trying to fix application-level design mistakes).

The MVEE we use includes the capability of taking snapshots of the state of each variant using CRIU[1] (Linux Checkpoint/Restore In Userspace). We analyze snapshots of the variants' state at the time of attack detection to identify the root cause of an attack. We recover from the attack by repairing the compromised system state in these CRIU images, then use the MVEE to reload the repaired state and resume execution as if the attack had never occurred (and without losing program state). Our approach provides state repair for of globals, stack, heap and registers as well as execution state for each variant.

[1]`https://criu.org/`

2 Methodology

We illustrate our approach on Apache prefork systems. Apache prefork is a non-threaded, pre-forking web server. Each server process listens for connections and serves them when they arrive and a parent process manages the size of the server pool and is responsible for launching child processes. Apache `httpd` always tries to maintain several spare or idle server processes, which stand ready to serve incoming requests. In this way, clients do not need to wait for a new child processes to be forked before their requests can be served. This design is useful for sites that need to avoid threading for compatibility with non-thread-safe libraries. Additionally, by isolating each request a problem with a single request will not affect any other.

2.1 Framework

To simplify the presentation, we assume an Apache prefork system runs two variants, each of which contains three processes serving requests. We produce two variants, where the order of globals and stack variables in one variant are laid out in the reverse order in the second variant.

Our framework identifies correlated processes across variants in the set, as shown in Fig. 1. Using the snapshot of the state of each variant, we pair corresponding processes and repair each pair independently.

2.2 Identifying An Attack

Consider a two-variant set, with seven global variables from A to F. The globals are arranged in opposite order in these variants. This means that an overflow past the end of one variable will corrupt different variables in each variant (see Fig. 2).

Variant-1 is compiled normally, shown in the upper half of Fig. 3. As given in the lower half of Fig. 3, in variant-2, the global variable allocation is forced to grow in the reversed direction against that of variant-1.

In this example, suppose that there occurs a buffer-overflow attack from the global variable B. The MVEE propagates any input to all variants identically. Therefore, an attacker's corruption via buffer overflow results in inconsistent states between the two variants because the attack corrupts the variable C in variant-1 but variable A in variant-2. Thus, by checking global

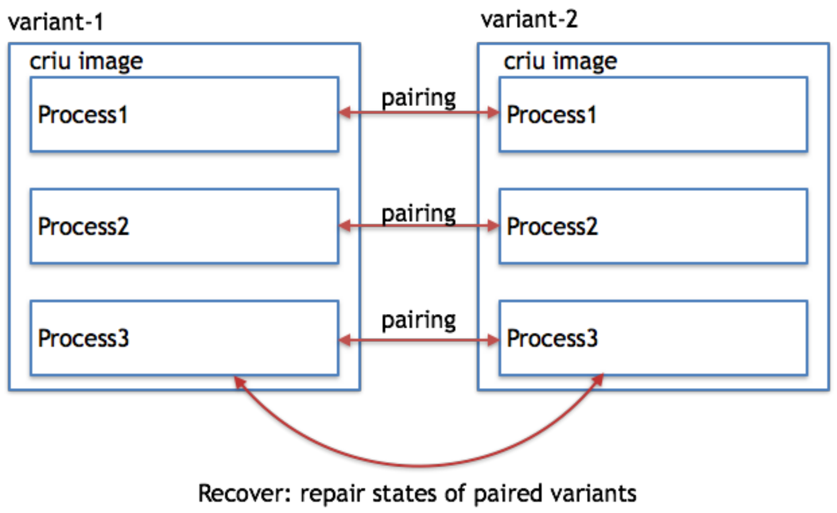


Figure 1: Our approach identifies and repairs corresponding pairs of processes across the variant set.

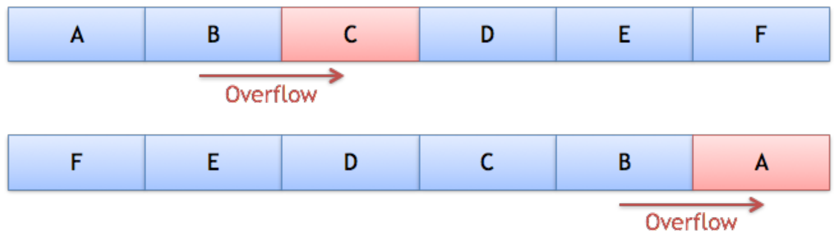


Figure 2: Two variants with reversed globals layouts. An overflow past the end of variable `B` corrupts different variables in each variant.

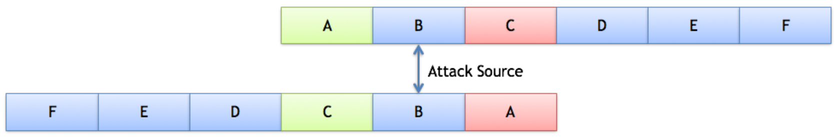


Figure 3: The attacker's payload will be identical across a multivariant set, allowing us leverage the reversed-layout pair to identify the attacker's attack source.

variable values in the available CRIU image, our analysis can detect the source of an attack, attack direction, and attack buffer size.

Specifically, in Fig. 3, we see that since the values of A and C are no longer

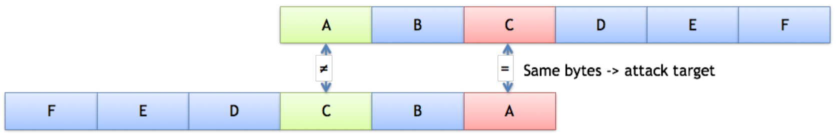


Figure 4: After identifying the source and extent of the attack, we determine the corrupted variables (red) and the corresponding uncorrupted values in opposite variants (green).

correlated after the attack, it is likely that there was an attack on the variable B. We show in Fig. 4 how our approach figures out the direction of the attack, *i.e.,* underrun or overrun. It checks the buffer content preceding and after the attack source B. Since the memory content spanning the variables B and C in variant-1 is equivalent to that of B and A in variant-2, we conclude that this is due to a buffer-overflow attack (a buffer-underrun attack would be in the reversed direction). By comparing how many bytes are actually equivalent, our analysis can detect the buffer size of the buffer-overflow attack. Our analysis is not sound in general as it can fail to identify the source of the attack when the global variables A and C had the same value before the attack. However, we've found our approach works quite well in practice in our experiments.

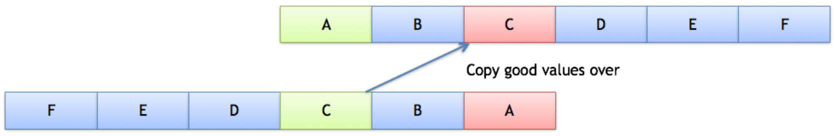


Figure 5: We repair the state of each variant using uncorrupted data from the opposite variant.

After identifying the source and direction of the attack, our analysis can repair the compromised state in the CRIU image. Fig. 5 gives the intuition. Since there was a buffer-overflow attack, variant-1 should hold the correct value of the global variable A while variant-2 should hold the correct value of B. It is straight-forward to use the value of C in variant-2 to construct the uncorrupted value for C in variant-1. Repairing the value of A in variant-2 follows the same principle.

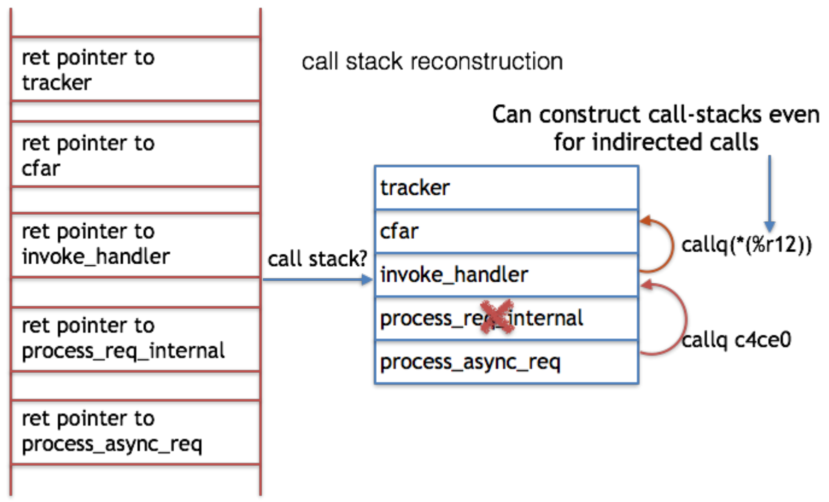

ret pointer to tracker
ret pointer to cfar
ret pointer to invoke_handler
ret pointer to process_req_internal
ret pointer to process_async_req
call stack reconstruction
Can construct call-stacks even for indirected calls
call stack?
tracker
cfar
invoke_handler
process_req_internal
process_async_req
callq(*(%r12))
callq c4ce0

Figure 6: Stack Frame Reconstruction: identify code pointers on the stack, then inspect program code to construct a feasible call stack (even those with indirect calls).

2.3 Stack Frame Reconstruction

Sometimes the MVEE does not detect divergence across variants until some time after the attacker's corruption of program state. In this case, even if we repair the program data (e.g., corrupted globals), the control flow of the variants may no longer be correlated due to control flow that depended on variables that were corrupted (or not) in each variant. In order to repair the state of all variants and allow the set to resume execution, we need some approach for also repairing the control flow state to once again correlate.

Our approach must clearly identify the call frames on the variants' stacks, *i.e.,* reconstructing the stack frames, to analyze where the two variants differ in terms of control flow. To this end, we designed a novel algorithm that can construct stack frames from a CRIU stack image.

Our frame reconstruction algorithm consists of two phases. In phase 1, given a CRIU snapshot of a stack segment, we scan the snapshot by reading values of every 8 bytes for finding all possible code pointers. Our system considers an 8-byte value a possible code pointer if that value falls in between the start address and end address of the code segment of the CRIU snapshot. We also determine which function a possible code pointer refers to by analyzing the relevant instructions. An example of this step is given in the left-hand side of Figure 6.

In phase 2, we construct stack frames from these code pointers. By observing the left-hand side of Figure 6, it looks like the referred functions form a sequence of call stacks.

$$\text{process\_async\_req} \rightarrow \text{process\_req\_internal} \rightarrow \text{invoke\_handler} \rightarrow \text{cfar} \rightarrow \text{tracker}$$

Here $x \rightarrow y$ means a x calls y. We use heuristics to identify potential code pointers, then perform additional validation that they do form a feasible call relation.

For example, we examine the instruction referenced by the return code pointer to `process_async_req` in the code of the `process_async_req` function (*i.e.,* the machine instructions in the binary containing the function). The code shows that the called function is `invoke_handler`. Therefore, in Fig 6, we have identified that `process_req_internal` is not a part of the call-frames that should be recovered. This solution works well but cannot deal with indirect calls. For example, in the right hand of Fig 6, the return code pointer to `invoke_handler` can be either form the stack frame of `cfar` or `tracker` because in the code of `invoke_handler`, whose address corresponds to this return code pointer, we see `callq(*(%r12))`. It is extremely difficult to guess what `r12` could point to without running the code. However, because we can confirm that `cfar` actually calls `tracker` from the code, we tentatively construct the following call-frames:

$$\text{process\_async\_req} \rightarrow \text{invoke\_handler} \rightarrow \text{cfar} \rightarrow \text{tracker}$$

This does depend on heuristics so our solution for stack frame reconstruction from a CRIU image is not precise due to the fact that we cannot soundly resolve indirected calls as in the case of Fig. 6. But, it has been sufficient for our experiments and testing.

When we identify call sequences from return code pointers, we simply reconstruct call frames whose upper and lower boundaries are defined by the stack addresses of such return code pointers.

2.4 Repairing An Attack

Comparing recovered stack frames between variants can reveal how control flow differs. For example, in Fig. 7, we show two variants that have very

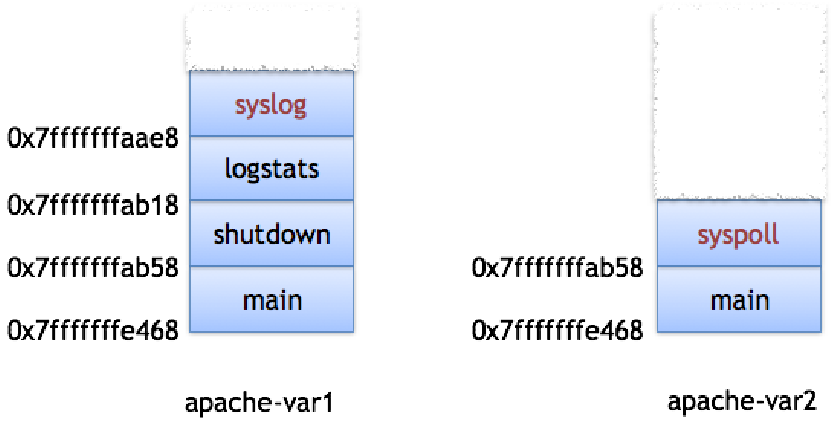


Figure 7: Example of two variants with different stack frames at the point divergence is detected after a global-variable overflow.

different call stacks, meaning that the control flow of the system began to diverge when executing the `main` function. We face two choices to make the system consistent shown in Fig. 8. We can copy the stack content from apache-var1 to that of apache-var2 or we can copy the stack content from apache-var2 to that of apache-var1. The question is which way should we choose. If this is not done carefully, after recovery, though the two variants are consistent, the attacker may gain the control of the system, *i.e.*, both the two variants have the control flow that the attacker wishes to enforce.

Our approach systematically investigates the root cause of a control-flow difference across variants. Our intuition is based on the observation that a control-flow difference must be due to a test (*e.g.,* `je`, `cmp`) on some variables that have different values across variants. These variable must have data-flow dependencies on variables that are compromised by an attack. Otherwise, there should not exist different control-flows. We developed a static analysis that tracks the dependency of a variable towards the set of variables that are known compromised by an attack. Specifically, our static analysis answers whether a variable is dependent upon a compromised variable. Our static analysis step symbolically executes the relevant machine instructions. A symbolic state contains a subset of the compromised variables that might influence the current state. Particularly, in a condition statement, if the branch predicate refers to a compromised variable, then the symbolic states along each branch contain this compromised variable because either of the

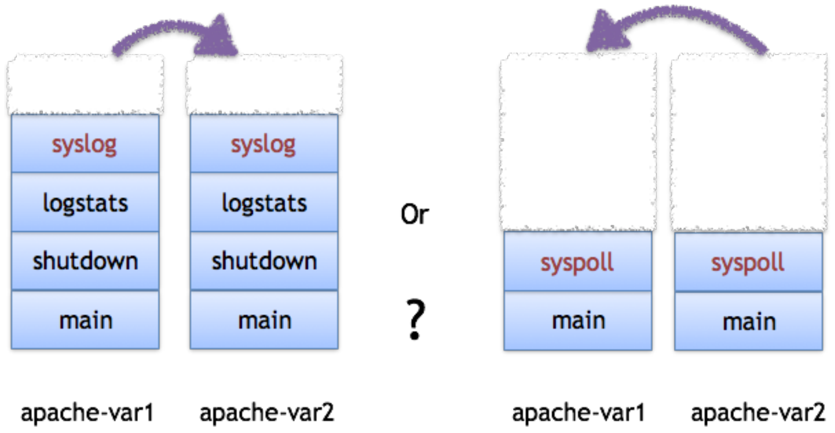
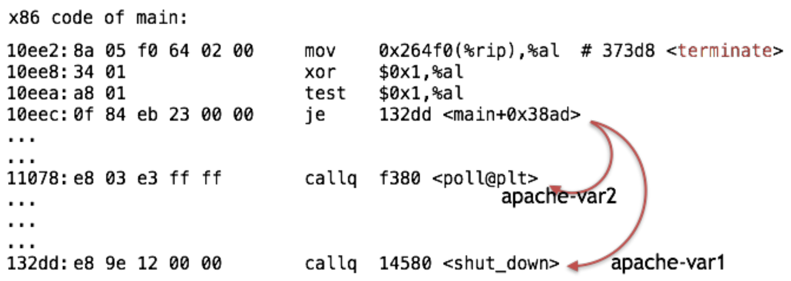


Figure 8: How to make the variants' states consistent? Should apache-var1's stack frame state be migrated to apache-var2, or vice versa?



```
x86 code of main:

10ee2: 8a 05 f0 64 02 00     mov    0x264f0(%rip),%al  # 373d8 <terminate>
10ee8: 34 01                 xor    $0x1,%al
10eea: a8 01                 test   $0x1,%al
10eec: 0f 84 eb 23 00 00     je     132dd <main+0x38ad>
...
...
11078: e8 03 e3 ff ff        callq  f380 <poll@plt>
...                                           apache-var2
...
...
132dd: e8 9e 12 00 00        callq  14580 <shut_down>      apache-var1
```

Figure 9: We perform lightweight symbolic execution to determine the correct stack frame state to restore throughout the variant set.

two branch execution clearly has a dependency on the variable.

For example, in Fig. 9, we study the code of the main function in Fig. 7. The variant apache-var1 makes the system call shut_down while the variant apache-var2 makes the system call poll. The symbolic states at these points generated by our static analysis include the compromised global variable terminate. Fig. 10 shows the scenario more clearly. The variant apache-var1 takes the true branch of the test on `test $0x1, terminate` because the value of terminate equals to 1 in its state. The variant apache-var2 takes the false branch of the test because the value of terminate is not equal to 1 in its state.

Consider the case where a two-variant set with reversed globals layout is

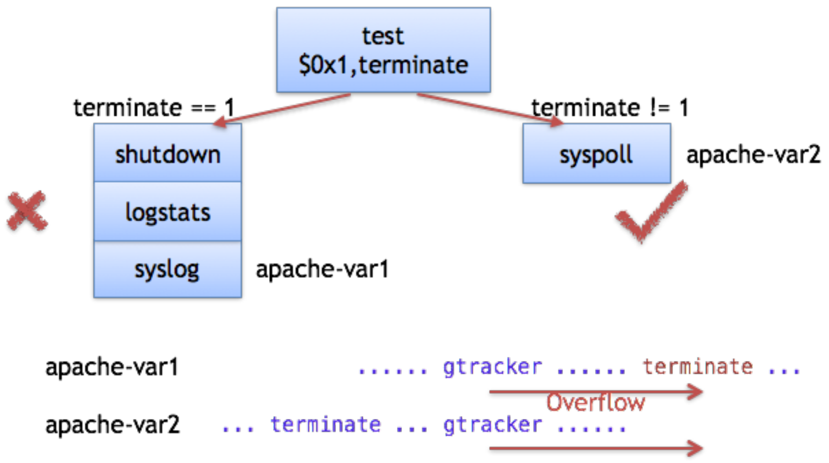


Figure 10: We identify the point of diverging control flow and which variant operated on uncorrupted state.

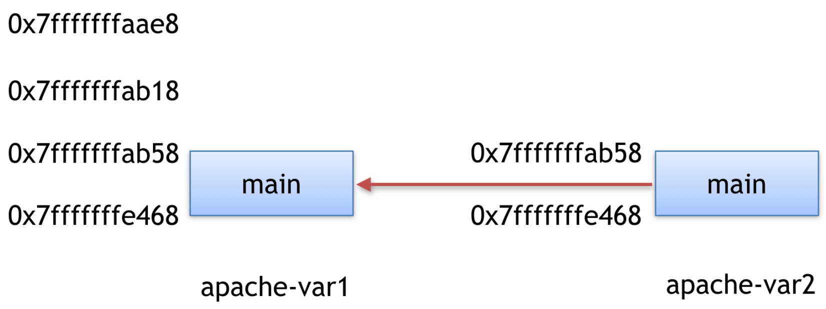


Figure 11: We repair the stack frame of the variant that diverged based on the use of corrupted data.

attacked with a buffer-overflow from the global variable `gtracker`, corrupting the global variable `terminate` in the variant apache-var1 but not apache-var2 (see Fig. 10). Therefore, the control flow in the variant apache-2 is the correct one while that of the variant apache-var1 is corrupted.

After identifying which variant still holds the desired control-flow structure, repairing stack content is straight-forward. For example, in Fig. 11, we copy the stack content from apache-var2 to that of the variant apache-var1. Importantly, we also need to reset the stack base pointer of apache-var1 using the stack base pointer value of apache-var2. We reset the `IP` pointer in apache-var1 similarly. Lastly, our system repairs inconsistent values in registers and heap. Since we already figured out which variant holds the correct control-flow, we copy the content of heap and registers from that variant to the other variant. During this process, we, however, need to track pointer values and make sure that they are meaningful. For example, copying a pointer to a global variable in the heap of apache-var2 to that of apache-var1 does not make sense because the global variable in apache-var1 has a different address since its global variable layout is reversed than that of apache-var2. We track the mapping relation between variables and their addresses in both variants and make sure that, when pointer values are copied from one to another, these values point to sensible variables.

3 Results

We implemented our state repair approach in a prototype in Java. It take as input CRIU images generated by a runtime monitor system based on multi-variant execution. It analyzes the root cause of an attack that triggered the dump of the CRIU images and repairs inconsistent state variables in the globals, registers, stack and heap regions in the images. The runtime monitor system can then resume its execution from the repaired CRIU images. We tested our state repair approach in Thttpd, an open source software web server and, Apache Prefork. We considered three classes of attacks in our experiments:

- a) Disclosure Attack – disclosing information such as code/variable address.
- b) Global Buffer Overflow/Offset Attack – injecting more data into a fixed-length buffer than the buffer can handle in global space.
- c) Stack Buffer Overflow Attack – injecting more data into a fixed-length buffer than the buffer can handle in stack space.

Our approach can repair damages made by all the above three kinds of attacks. From our repaired states, a Thttpd or Apache Prefork server system can resume successfully. The repair time is negligible – typically less than 2 or 3 seconds. Our experiments demonstrated that it is possible to perform state repair to recover from common forms of memory corruption attacks.