Reimplementation of ConVul, a Memory Corruption Vulnerability Detector

1st Kenneth Lamar
Department of Computer Science
University of Central Florida
Orlando, United States
kenneth@knights.ucf.edu

2nd Kimberly Stevens
Department of Computer Science
University of Central Florida
Orlando, United States
k.stevens@knights.ucf.edu

***Abstract*—Concurrency vulnerabilities cause great damage yet are notoriously difficult to identify. Non-deterministic thread interleavings hide bugs that only occur with specific thread timings, which attackers can leverage. Checking all possible interleavings is computationally infeasible on all but the simplest programs. There are existing works that implement various data race detectors to detect concurrency vulnerabilities, but these methods have proven ineffective; many data races are benign while others cannot be effectively leveraged into an attack. Rather than using race detection, a recently provided tool, ConVul, is designed to detect use-after-free (UAF), double-free (DF), and null-pointer-dereference (NPD) vulnerabilities made possible by thread non-determinism. These commonplace memory vulnerabilities are easier for an attacker to leverage than race conditions and identify a class of bugs that traditional race detectors cannot detect. Detection is accomplished using a trace from a single run, alongside a thread causality model, to identify many feasible traces through `lock()-unlock()` blocks. Experimental results for ConVul show state-of-the-art memory vulnerability detection, yet no source code is available. We have identified this gap, reimplemented the tool based on the author-provided paper, and compared our design against theirs on the same test suite as the original work. In testing, our implementation appears comparable in most cases, identifying seven out of ten CVEs while ConVul identifies nine out of ten.**

***Index Terms*—concurrency, data structures, security**

I. Introduction

Concurrency vulnerabilities cause great damage yet are notoriously difficult to identify. These vulnerabilities exist in multi-threaded programs due to specific thread timings and have proven to be difficult to detect. One example of such a vulnerability is DirtyCow in the Linux kernel. When DirtyCow is taken advantage of, a normal user can gain root privileges. Clearly, it is desired to avoid this kind of vulnerability. Most attempts to find concurrency bugs use data race detectors. However, these methods have proven ineffective; many data races are benign while others cannot be effectively leveraged into an attack. A focus instead on memory allocation vulnerabilities caused by non-deterministic thread interleaving is more promising, as these bugs can be more easily leveraged into attacks.

There are patterns and certain primitives to search for to find the possible occurrence of a concurrency bug. While a sequential program can be evaluated by analyzing the program in order, a multithreaded application requires the evaluation of an exponential number of possible instruction interleavings. To identify every possible pattern requires checking every possible thread interleaving, which is computationally infeasible. This is sometimes referred to as the interleaving space explosion problem.

An alternative approach is to use a trace from a single run of the program alongside a thread causality model to identify many feasible traces. This is an inherently approximate approach to vulnerability detection, but it makes analysis computationally feasible even on large projects. In previous work, the maximal causal model has been used, though it has been found to miss real bugs. Because of this, a more relaxed model is often used. ConVul is one application of a relaxed model, providing a useful vulnerability detector. This tool has impressive experimentation results and a great success rate of discovering new vulnerabilities. In the testing of their tool, ConVul found nine out

of ten concurrency vulnerabilities and six zero-day vulnerabilities, four of which have been confirmed. CONVUL does this by finding events that affect the same memory blocks that may be exchangeable, or happen in any order [1], [2]. Exchangeable events are sometimes defined by synchronizations, such as when they are used with data races. Here, however, exchangeable events are defined across synchronizations. This differentiates and broadens the scope of CONVUL compared to data race detectors. These exchangeable events are where concurrency vulnerabilities will occur, so the vulnerability can be reported to the user. CONVUL successfully identifies use-after-free (UAF), double-free (DF), and null-pointer-dereference (NPD) vulnerabilities [2]. However, CONVUL, has no available source code. We feel that this is an important tool and many would benefit from having it available for use. Because this tool would be beneficial, we have chosen to reimplement it using the documentation the CONVUL team provided as part of their paper.

In summary, this research work provides:

- A description of CONVUL and our reimplementation of the tool.
- Source code for our implementation.
- An accuracy and overhead comparison between the original implementation and our reimplementation.

II. RELATED WORK

Before discussing the reimplementation and the design choices made, we will first describe the original tool that we were trying to reimplement. This section is a summary of the original CONVUL tool as described in the documentation found in [1], [2].

A. CONVUL *Background*

CONVUL is a pintool created using Intel Pin [3] to work with C/C++ programs on the Linux Operating System. This tool focuses on use with multi-threaded programs to identify use-after-free, double-free, and null-pointer-dereference vulnerabilities. To do this, CONVUL has to track different events and determine relations between them.

B. *Event Tracking*

There are several kinds of events CONVUL looks for. The first is a memory event. This is either a thread writing to memory or a thread reading from memory. The second is a synchronization event. This is a lock acquisition by a thread or a lock release by a thread. Finally, memory free events are tracked to determine when portions of heap memory are freed. A trace is created as events occur, and there is a trace for every thread to track all the events performed by that thread. Tracing provides a natural sequence of events by each thread, but we also need to consider sequences of events between threads, as this is where most concurrency bugs appear.

CONVUL uses the happens-before relation (HBR) to help define the ordering of events between threads. Vector clocks (VCs) are used to track HBR for different events. VC algorithms are considered well-known by the original authors and are not described in detail.

Not all events in a multi-threaded program are well-defined by HBR. Sometimes, two events may occur in either order. In other words, there is no synchronization mechanism preventing the second event from happening before the first event. This is oftentimes where a concurrency bug can be found and leads us to the idea of exchangeable events.

Exchangeable events are any two events in which either ordering of those events may feasibly be observed. It is uncommon to view both orderings in a limited number of runs, so the CONVUL team designed a way to determine if two events are exchangeable based on the number of sync edges between the events. A sync edge is best described considering a program similar to a graph where the events are nodes and edges are lock releases and lock acquisitions. A sync edge runs either (1) from a lock acquisition to a lock release on the same thread, or (2) from a lock release to a lock acquisition on different threads. The fewer sync edges that occur between the two events in question, the more likely it is that they can be exchanged. Due to this realization, CONVUL only considers events that have three or fewer sync blocks between them. This is considered 3-exchangeable. While other distances were tested, the authors found this approach to be ideal, as larger distances were not more effective yet

increased evaluation time.

C. Check Exchangeable Events

As stated previously, VCs help track HBR between events. For every event that occurs, a prediction is made based on the VC of the thread where the event occurred. This prediction is another VC that may be used for comparison later. To actually check if two events are exchangeable, one must first check for a clear HBR. If no HBR exists, these events are exchangeable. If a HBR is found, a more involved check of the thread trace is necessary to consider longer sync distances. Three events will be taken into account here: a lock release(e_1), a lock acquisition(e_2), and finally the event that occurs immediately before the acquisition(e_{any}). This means that at most one event is between e_{any} and e_2. If e_1 and e_{any} do not have a clear HBR and the VC of the lock is the same as the already predicted VC of e_1, then these two events are exchangeable. Any other circumstances means the events are not exchangeable.

D. Detect Concurrency UAFs

The first algorithm we will describe is to detect the use-after-free(UAF) vulnerability. This occurs where one event is accessing memory after another event has already called `free()` on the same block of memory. To test this, we need to update the VC of the event that is accessing the memory block with the VC of the thread where the event is occurring. This VC can be used later if it is possible that this event is exchangeable with another. Now that every event accessing memory has a VC, every time an event attempts to free the memory block, another algorithm is used. This algorithm will first get the size of the memory block that the event is trying to free. Then for each index of this memory block, check for events that have already occurred which accessed this place in memory. If any of the events found are exchangeable with the free event, this algorithm will report a UAF. In summary, the algorithm tracks every access made to memory and checks each event for exchangeability, as described above, with the free event of that memory block.

E. Detect Concurrency NPDs

The next algorithm describes how to detect null-pointer-dereference (NPD) bugs. This vulnerability occurs when a pointer was set to NULL, then later is dereferenced. Unfortunately, this bug is less straightforward than the last because one or more events can occur between the set to NULL and the dereference. Three distinct cases will be described in addition to the trivial case.

The trivial case is where two threads are involved. The first is meant to dereference the pointer and the second is meant to set the pointer to NULL, but due to thread timings, these two events can be reversed.

The next three cases will assume that some third event occurs, let this event be writing to the pointer, such as $p = q$. The first of the non-trivial cases is where the second thread that sets the pointer to NULL first writes to the pointer. Then after the pointer is set to NULL, the first thread dereferences the same pointer. The second of the non-trivial cases occurs when the second thread sets the pointer to NULL; next, the first thread attempts to dereference the pointer; then also tries to write to the pointer after the dereference. The final case is where a third thread is introduced. This case starts the same as the second case, but after the pointer is dereferenced, the third thread is responsible for trying to write to the pointer.

The algorithm to check for NPDs has to track all writes to a pointer, both NULL and Non-NULL, and all reads, or dereferences, to a pointer. To do this successfully, three additional VCs for pointers are used one for each event: reads, NULL writes, and Non-NULL writes.

There are two functions that help determine NPDs, the first is called for each write event. First, determine if the write has a value of NULL. If so, for any read events, check if it is exchangeable with this write of NULL. If the two events exist and are exchangeable this is an example of the trivial case, report the NPD. Finally, update the appropriate VC for the write event with the VC of the thread where this event is occurring.

The second function is called on the memory read events. First, determine if the read event is a dereference of a pointer. If it is not, return from the function call otherwise continue. If this is a dereference instruction, check for all write events to the same pointer, both NULL and Non-NULL values. If any Non-NULL write events are exchangeable with any NULL write events this is an example of

the first non-trivial case, report the NPD. Otherwise, check if the read event is exchangeable with any of the Non-NULL write events. If so, this is an instance of either the second or third non-trivial case, and a NPD is reported. If not, this is the last possibility and there is no NPD. Update the read VC with the VC of the thread where the read event is occurring.

F. Detect Concurrency DFs

A double-free vulnerability occurs when a program attempts to free the same block of memory twice. This can be hidden when one or more events occurs in between the two free events. Similar to that of the NPD vulnerability, there is a trivial case as well as three non-trivial cases.

The trivial case is where two different threads call free on the same memory location due to poor synchronization. The non-trivial cases occur when a thread writes to the pointer in between the two free events. This can be the thread of (1) the first free, (2) the thread of the second free, or (3) some third thread. If the thread timings are correct, then the second free event is actually referencing a different block of memory after the event in between the free events occurs, so this would not be a DF. However, if the thread timings are off and these events occur out of order, then it may result in a DF.

This algorithm is tricky because the pointer accesses and modifications need to be tracked carefully. This is accomplished using maps for each memory location. First, the map *Pts* is used to map a memory location to a set of pointers to that memory location. In other words, it lists all pointers that point to the current memory location. Second, the map *Frs* is used to track all events that free the current memory location, which is a pointer. Once again, two functions will be discussed to detect concurrency DFs.

The first function is called any time a pointer is assigned. This function is how *Pts* is updated and tracked. When an address is assigned to a pointer, remove all items that are currently mapped to that to that pointer from other memory blocks, then update the pointer to be mapped from that memory block.

The second function is called on every free event. For every pointer that exists in *Pts*, if a free event exists in *Frs*, check if the free event is exchangeable with the write event associating the pointer to the memory location. If they are exchangeable report a DF. Otherwise check if the same write event is exchangeable with the free event that called the function. If these two are exchangeable, report a DF. If not, there is no DF here and *Frs* must be updated.

III. Reimplementation Design Differences

This section provides a design overview of our reimplementation. Fundamentally, our design is closely based on the provided ConVul design. Some changes were necessary to handle design ambiguities, while additional improvements were made by applying performance optimizations or simplifying the design when compared against the original pseudocode.

In the original design, VCs were tracked and seemingly associated with various objects, such as memory trackers, within the instrumentation code. Most of these VCs were not explicitly specified, deferring to a paper that uses VCs to identify HBR violations instead [4]. Ultimately, we found that the most straightforward correct approach would be to associate a VC with each thread and each lock. Every time a thread performs a tracked event, it increments its associated counter in its own VC. Whenever a lock is acquired by a thread, the lock updates the thread's VC with any newer clock counters. Whenever a lock is released by a thread, the thread updates the lock's VC with any newer clock counters. In this way, VCs model a lock's enforcement of operation ordering between threads. Only if each index of the VC is less than or equal to the corresponding indexes of another VC does it have a clear HBR. Unlike the paper, which seems to have specialized VCs for specific event types, we track VCs for every event. Whenever a thread performs a tracked event, the event gets a copy of the thread's VC at the time it performs the operation. This makes it easy to track HBR and exchangeability between any two events, using the algorithms described in the original paper.

ConVul needs a way to calculate the sync edge distance between two events to determine whether the events are exchangeable. In the authors' implementation, they used an optimized approach that, rather than using a graph to track sync edge distances, uses some clever shortcuts that work well

when checking for distances less than or equal to three. The algorithm starts by checking for a HBR between the two events. If the events appear concurrent from this check, then they are exchangeable. This will occur if the sync edge distance is equal to zero. Checking for distances as high as three in the authors' optimized approach involves the following events, in execution order: (1) the earlier of our two compared events, e_1; (2) e_{rel}, an event that releases the lock held by the thread performing e_1; (3) e_{any}, an arbitrary event that occurs before the lock associated with e_2 and is run and on the same thread as $e2$; (4) e_{acq}, an event that acquires the lock before e_2; and (5) e_2, the later of our two compared events. The sync edge distance is less than or equal to three if we can find that e_{any} and e_1 do not have a clear HBR with each other and that the lock associated with e_{rel} and e_{acq} are the same lock. To ensure they are the same lock, the authors use an approach where each event tracks a predicted VC for each event. If the lock is the same, then the VC of the lock should be the same as the predicted VC during the evaluation.

Our approach to exchangeable events is similar, but we simplify our design through the use of both our per-event VCs and the use of tracked thread blocks. A block tracks information associated with a given instance of a thread holding a lock. Whenever a thread acquires a lock, it creates an associated block entry storing the trace index of the lock acquisition event and a pointer to the tracked lock itself. Any event that occurs while the thread holds the lock gets a copy of each block so it can be associated with the held lock. When the thread is released, the trace index of the releasing event is set in the block. The block object makes it easier to perform the optimized approach described by the original paper. e_1 and e_2 are provided for exchangeability searches, but the approach described in the paper requires extensive trace searching to locate the remaining three events associated with an exchangeability check. Using blocks means all of our events are readily accessible. The e_{rel} event can be found by checking e_1's associated block for the releasing event. The e_{acq} event can be found by checking e_2's associated block for the acquiring event. The e_{any} event can be found by accessing the event in the trace occurring just before e_{acq}. Finally, we do not need to compare the lock's VC against a predicted VC to verify that e_{rel} and e_{acq} are the same lock. Instead, we can simply check the blocks associated with e_1 and e_2 to ensure they reference the same lock. Finally, the authors never discuss how to approach exchangeability checks when more than one lock is involved. We can quickly iterate over the block list for e_1 and e_2 to identify shared locks and quickly perform the check on each lock.

When a program is concurrent, Pin analysis will also run concurrently. Thus, analysis must be protected by synchronization primitives, such as locks. The authors do not discuss their use of locks, even though this aspect of design is critical for both correctness and performance. We use a relatively coarse approach to synchronization, with one lock for the entire set of tracked memory locations, one lock to protect access to the lock map, and an instrumentation lock for each actual lock. Per-thread tracked data uses no locks, as this data is effectively thread-local, accessed only by the associated thread. While this coarse approach may be responsible for low performance, it was more important for our implementation to work correctly.

When detecting a DF, pointer assignment requires iterating over every tracked memory location to remove the modified pointer from their pointer list, even though only a single location will be pointed to at a given time. We found that this operation could be sped up if each pointer instead logs the last event to change its value and each event stores the value written. This way, the stored pointer value can be used as the address of the specific memory location to check for and remove the pointer from the list, making a formerly $O(n)$ operation run in constant time. There were many other places in the pseudocode that required finding the existence of an event that meets specific constraints. Naively, one may iterate over all of the traces, but we could often accelerate look ups via maps. We use map data structures to associate memory addresses to each memory instrumentation object and associate lock addresses with each lock instrumentation object.

IV. Experimental Results

A. Overview of Benchmarks

The benchmarks used for testing ConVul were a variety of well-documented concurrency vulnerabil-

CVE	Actual Bug	Original	Ours
2009-3547	NPD	✓	✓
2011-2183	NPD	×	×
2013-1792	NPD	✓	✓
2015-7550	NPD	✓	✓
2016-1972	UAF	✓	✓
2016-1973	UAF	✓	×
2016-7911	NPD	✓	×
2016-9806	DF	✓	✓
2017-6346	UAF (DF)	✓	✓
2017-15265	UAF	✓	✓

TABLE I
DETECTION RESULTS.

ities. According to [1], [2] there were a total of ten CVEs used, including five NPDs, four UAFs, and one DF. CONVUL successfully identified nine of the ten vulnerabilities. The missed benchmark was cve-2011-2183. This specific NPD was outside the scope of CONVUL because the vulnerability is generated by calls that are made by the same thread as stated by [2]. Compared to data race detectors that were also tested against the same CVE set, CONVUL found far more bugs than any of the detectors. Each detector only found one or two of these known vulnerabilities.

In addition to the CVEs, CONVUL was also run against the MySQL database server. CONVUL detected six zero-day vulnerabilities on the MySQL database server, four of which have been confirmed. These bugs include five NPDs and one UAF. Two of the NPDs have yet to be confirmed [1], [2]. In comparison, the same data race detectors used with the CVEs were tested against the MySQL database server. Each of the related works compared against found no more than one of the vulnerabilities.

B. Reimplementation Effectiveness

We used CVE tests provided by the original authors to compare our design against theirs. As shown in Table I, our reimplementation found seven of the ten CVEs, while the original design finds nine of the ten. The reason for this discrepancy is not fully understood, but there are some additional approximations in place that will make our results inherently different than the original.

First, our approach does not filter out memory addresses based on stack size for each thread, meaning some additional false positives may be identified. Second, when checking for UAF, a free event should check each byte associated with the object to ensure overlapping memory accesses are recognized. We initially used `malloc_usable_size()` to calculate this, but it often returned large values that slowed down analysis to take hours instead of seconds. As an approximation, we decided instead to just check the base address itself instead of all possible offsets within the allocated size. Finally, the authors claim to use some form of type checking to distinguish pointers from other types. For instance, this allows assignment of a value of 0 to be ignored rather than treated as an assignment of NULL. Type checking in Pin is not natively supported, and it is unclear how they determined this reliably. It appears that they may have accomplished this using their `isDeref()` function, which is a heuristic to guess if a memory read is a dereference. This allows non-reference reads to ignore NPD analysis code. Finally, since each run of the tool results in a single, non-deterministic interleaving of operations, it is possible that the CONVUL authors encountered slightly different executions that resulted in different coverage of bugs than our implementation.

The authors claim that CONVUL never reports any false positives due to a validation step, which verifies the existence of a real vulnerability by enforcing the suspect interleaving, executing it, and checking to see if a crash occurs. This validation process is not detailed in the paper. Evaluation of the provided CVE tests suggest that some of the tests may have had this done manually, using an additional lock added to the code to artificially stall a specific thread. In our implementation, we do not perform any validation, meaning many false positives may be present. Unlike the authors' original design, we found that our reports' instruction traces often had difficulty locating symbols for meaningful file name and line number information, making verification that our detection matches difficult.

In the original paper, CONVUL reports an average time overhead of 269.8x and an average memory overhead of 81.0x. In our reimplementation, we found an average time overhead of 4380.0x and and average memory overhead of 684.7x. These sig-

nificant differences suggest that several aggressive optimizations were applied but left undetailed in ConVul. It is possible that they measured metrics differently than we did. We recorded time and memory usage using the Linux `time` command, reporting the wall clock time and maximum resident set size. It's also possible that they timed less of the execution, as we timed entire tool runs from beginning to end.

V. Conclusions

In this work, we developed the first publicly available implementation of ConVul. While it has some limitations when compared against the original tool, it still provides useful reports on memory access violations that can lead to real concurrency attacks. In future work, we hope to improve the framework laid out by our reimplementation to reach feature and performance parity with ConVul. Additionally, we would like to extend this work to support vulnerability detection on non-blocking atomic primitives, particularly when used by atomic data structures. We believe that the well-defined correctness conditions of these data structures can be applied to identify feasible reorderings from a single trace, even in the absence of locks.

The codebase for our reimplementation can be found at https://github.com/KennethLamar/ConVul.

References

[1] R. Meng, B. Zhu, H. Yun, H. Li, Y. Cai, and Z. Yang, "Convul: An effective tool for detecting concurrency vulnerabilities," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1154–1157.

[2] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, "Detecting concurrency memory corruption vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 706–717. [Online]. Available: https://doi.org/10.1145/3338906.3338927

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[4] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.