The BORG: Nanoprobing Binaries for Buffer Overreads

Matthias Neugschwandtner
Vienna University of Technology
SBA Research

Paolo Milani Comparetti
Lastline Inc.

Istvan Haller
VU University Amsterdam

Herbert Bos
VU University Amsterdam

ABSTRACT

Automated program testing tools typically try to explore, and cover, as much of a tested program as possible, while attempting to trigger and detect bugs. An alternative and complementary approach can be to first select a specific part of a program that may be subject to a specific class of bug, and then narrowly focus exploration towards program paths that could trigger such a bug.

In this work, we introduce the BORG (Buffer Over-Read Guard), a testing tool that uses static and dynamic program analysis, taint propagation and symbolic execution to detect buffer overread bugs in real-world programs. BORG works by first selecting buffer accesses that could lead to an overread and then guiding symbolic execution towards those accesses along program paths that could actually lead to an overread. BORG operates on binaries and does not require source code. To demonstrate BORG's effectiveness, we use it to detect overreads in six complex server applications and libraries, including lighttpd, FFmpeg and ClamAV.

1. INTRODUCTION

Buffer overreads are an increasing security concern in modern computer systems. Virtually all of today's major operating systems employ advanced protection mechanisms like Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) to prevent attackers from diverting the control flow of a program to executable code. Because code addresses are no longer easily guessable, attackers rely on memory disclosures such as buffer overreads to find them. Typically, even a single disclosure suffices to bypass even these powerful defenses [36, 37]. Besides addresses, buffer overreads may leak sensitive user information and lead to crashes. A very recent example is the Heartbleed bug in OpenSSL [14] that allows exfiltration of cleartext data.

Even so, the problem of overreads in binaries has received little attention by the research community. The problem is that finding buffer overread vulnerabilities is hard enough if the program's source code is available, but without the source or debug symbols, it is almost impossible. Unfortunately, most commercial software consists of optimized, stripped binaries. Analysis of binary programs is desirable for developers that rely on compiled third-party COTS libraries or programs as well as security analysts auditing binary programs.

In this work, we introduce the BORG (Buffer Over-Read Guard), a tool for finding buffer overread bugs with guided symbolic execution. BORG is based on S2E [12] and works on binaries, with no source code required. At the core of our system are novel techniques for guiding the execution to a potentially vulnerable access while at the same time trying to trigger an overread on this access. To the best of our knowledge, BORG is the first automated bug finding tool targeted at finding buffer overread bugs, so we believe it can be useful in practice to harden programs against memory disclosure vulnerabilities.

The BORG's guided symbolic execution. Symbolic execution is a powerful technique for finding bugs in software. By executing a program under symbolic inputs and forking execution to explore many different program paths, symbolic execution can automatically find bugs, as well as provide concrete input values that trigger them. The fundamental limitation of this approach, of course, is that it is infeasible to explore all possible program paths. Symbolic execution suffers from "path explosion", because the number of program paths to explore grows exponentially with the number of branch points encountered. This problem is compounded by the computational cost of symbolic execution: At each branch point, a symbolic executor will invoke a costly constraint solving step to decide which branches can be reached. This difficult problem gets even harder when source code is not available and the solver must reason on binary code. Furthermore, keeping track of program state for all of the as yet unexplored paths puts pressure on memory, further impacting performance.

Whenever symbolic execution reaches a branch point, we must decide which branch to execute first. The strategy used to select paths to execute is essential to the effectiveness of symbolic execution. For testing code and finding bugs, the goal of the path selection strategy is typically to improve the overall code coverage of symbolic testing. To this end, for instance, KLEE [8] alternates between a code-coverage based path selection heuristic and random selection.

Instead of trying to cover as much code as possible, a different approach is to guide execution towards "interesting" parts of the program under test. This is the approach we follow in this work. The general idea is to first select specific parts of the tested program that are more likely to be subject to specific bug classes, in our cases overreads, and to then use a path selection strategy that guides execution towards these interesting parts while trying to violate an integrity assumption.

BORG uses such guided symbolic execution to steer the program exploration toward code that may allow for overreads. Beyond the concrete implementation for overread detection, however, we will

CODASPY'15 March 02 - 04 2015, San Antonio, TX, USA
Copyright 2015 ACM 978-1-4503-3191-3/15/03 ...$15
http://dx.doi.org/10.1145/2699026.2699098.

argue that the proposed approach is general and can be applied to several classes of bugs, so long as we are able to:

1. Select potential targets in the tested program that are more likely to be vulnerable to bugs of a certain class.
2. Guide execution towards those targets.
3. Detect the occurrence of a bug as the violation of an integrity constraint.

The techniques we introduce for item 2 (guidance) are quite general. Furthermore, there are well-understood ways to express many classes of bugs as the violation of an integrity constraint (item 3). Therefore, the applicability of the techniques proposed in this paper to different classes of bugs is restricted chiefly by item 1: That is, by our ability to select targets that have a high enough likelihood of being subject to such bugs.

Contributions. To summarize, the contributions of this work are the following:

- We develop novel techniques for guiding symbolic testing of a program towards a potentially vulnerable target while trying to break an integrity constraint.
- We build a system called BORG that combines these techniques with a selection heuristic and a detection mechanism for potential buffer overreads to automatically discover such vulnerabilities.
- All of the techniques we introduce work on binary code and do not require source code.
- We apply BORG to six complex, real-world programs including lighttpd, FFmpeg, and ClamAV, and show that our system is able to automatically trigger buffer overreads in these programs.

2. OVERVIEW

Figure 1 shows a high-level overview of BORG. Given a binary program and test data as an input, BORG will set out to find an execution path that violates a security assumption and output the corresponding malformed input. The core of our tool is an online symbolic execution engine that runs the program and generates new states on every conditional control-flow branch that depends on the program's input. Three main components that build around and on top of this core principle are essential to BORG's functionality:

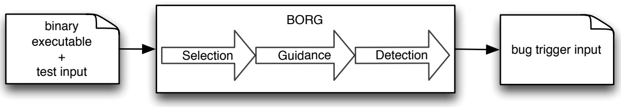

Figure 1: High-level overview of BORG

Selection. Given a specific integrity assumption, the goal of this first step is to identify potentially vulnerable spots, i.e. security critical code regions that are likely to violate the integrity assumption.

Guidance. The purpose of the guidance mechanism is to select the most promising states produced by the symbolic execution engine for exploration. Promising states are more likely to lead to a violation of the integrity assumption. Therefore the states are ranked based on their likelihood to (1) actually hit the potentially vulnerable spot and (2) allow actually triggering the bug based on the state's path constraints. To this end the guidance regularly assesses the states' properties and picks the state that is ranked highest.

Detection. While the guidance attempts to trigger a bug, the detection mechanism has to check whether the integrity assumption is violated. In case of a violation BORG will terminate the current state and output a test case for this execution path.

While the principles outlined so far are fairly general, our implementation of BORG focuses on detecting overread bugs. We will discuss additional classes of bugs to which our approach could be applicable in Section 8.

2.1 Testing Process

Figure 2 shows an overview of the steps and components involved in BORG's testing process. As input, BORG will accept a binary executable along with test inputs. To obtain inputs for the tested programs, we can use inputs in the program's test suite, if available. If the program itself does not have a test suite, provided that we know at least which kind of protocol or data format the program will handle, we can use test cases from existing test suites for the corresponding protocol or data format. Concrete examples from our evaluation are the HTTP protocol and the JPG file format. If on the other hand the expected input format is unknown, we can start from any real-world input data.

Preliminary Analysis. The preliminary analysis stage combines static and dynamic analysis to gather information about the tested program. First, we perform an instrumented, concrete execution of the tested program for each available test input. During this execution, we use dynamic taint analysis to propagate taint from the test input. We thus generate detailed execution traces that include taint information for instruction operands.

The first goal of the preliminary analysis phase is to generate an accurate model of the program's intra- and inter-procedural Control Flow Graph (CFG). For this, we refine a CFG obtained from static analysis with knowledge from the dynamically generated execution traces. The dynamic execution provides information that is not easily available from static analysis, such as possible targets of indirect control flow transfers.

We further use the execution traces to collect buffer access profiles that will later be used to detect out-of-bounds buffer accesses. Specifically, we record which buffers in the program's memory are accessed by which of the program's instructions.

Target Selection. After the preliminary analysis has been completed, the next step is to select targets for testing. That is, we aim to select specific instructions in the program under test that are more likely to trigger a bug, so we can guide our testing towards those instructions. For the detection of buffer overread bugs, BORG selects all memory reads from a tainted address: That is, from an address that is derived from the program's input. We also target sensitive functions such as `memcpy` and `strcpy` if their parameters are tainted. It is easy to see that whenever a buffer access's address depends on data that can be influenced by an attacker, there is a possibility that the access will go out of bounds if we can find a program path along which the address is not calculated or sanitized correctly. For this reason, these accesses are good candidates for targeted testing. As we will show, this selection strategy is already effective for finding non-trivial bugs in real programs. On the other hand, we must point out that it is not on its own sufficient to target all accesses that can lead to overreads, because overreads can also happen with no direct data flow from user inputs to addresses. Additional selection strategies for potential overreads could be employed alongside this one to obtain additional targets and find more bugs: one approach is to rank accesses in loops based on some measure of the complexity of the code that computes the accessed address [23].

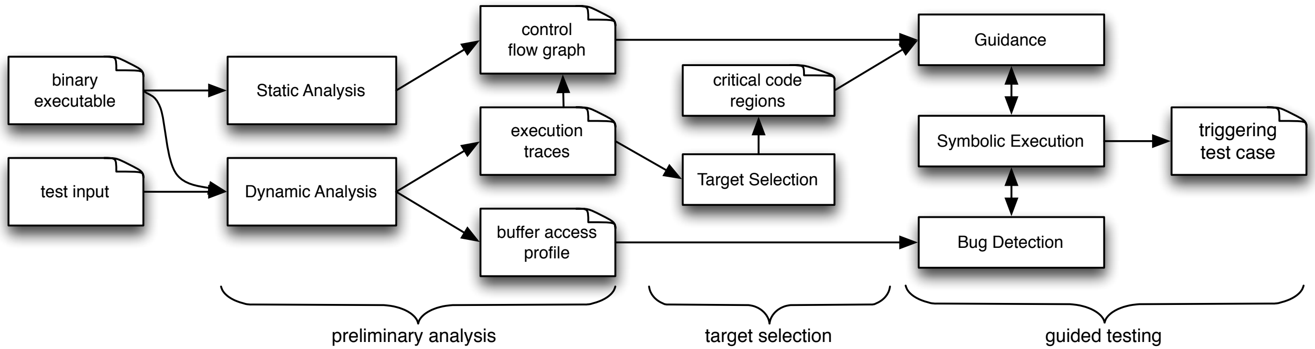

Figure 2: Overview of the testing process executed by BORG. The preliminary analysis applies both static and dynamic analysis to the binary program under test to produce information that is required for the components of the main testing phase. After the critical code regions have been identified based on the dynamic execution traces, guidance can use those in combination with control flow data to home in on the targets. At the same time, the detection component uses the buffer access profiles to detect overreads.

Guided Testing. Once the targets have been identified, we can begin to test the program using guided symbolic execution. We perform the guided testing step separately for each identified target. We consider as symbolic the entire program input (such as a video file for *FFmpeg*, or the input from a network socket for the *lighttpd* web server). Since we use an online symbolic executor, states that capture the complete current state of the program executed are forked whenever the execution encounters a branch point where the branch condition is based on symbolic input.

Two components constantly interact with the symbolic executor: guidance and detection. The guidance regularly assesses a number of characteristics of each state to establish a ranking that corresponds to the likelihood of triggering a bug. The symbolic executor will always execute the state with the highest rank. Intuitively, the ranking is designed to prefer states that are near to the target in the program's control flow graph and have path constraints compatible with the violation of the integrity assumption. Furthermore, the ranking uses additional heuristics aimed at privileging execution paths that are more likely to lead to an overread. Section 3 explains the guidance mechanisms in detail.

To find a bug, in addition to triggering its occurrence, we also need to detect that it was indeed triggered. Detecting a bug is not as straightforward as executing the faulty code since silent failures are typical for buffer overreads. To detect buffer overreads we have developed a technique that is effective for memory accesses that use either concrete or symbolic addresses. In some cases, detecting an overread may be trivial. As an example, if the address of a memory access can be made to point outside the program's allocated memory pages, it is obvious that an overread is possible. To detect a wider class of overreads, however, we need more fine-grained detection methods that are aware of buffer boundaries within the program's memory space. We will discuss in Section 4 how we obtain this information about a binary. To detect overreads, we make use of the buffer access profiles collected during the preliminary analysis. This profile associates each program state with the buffers accessed in that state during any of the the test runs in the preliminary analysis phase. Here, a program state is a combination of an instruction with the current call stack. During symbolic execution we again check, for each memory access, whether a buffer is being accessed. We then compare this information with the profile for the current program state. Based on this comparison we distinguish three conditions:

Valid access. If the access hits a buffer that has been recorded before in the current state's profile, the access is clearly valid. Buffer accesses in code that have not been covered by the preliminary analysis (for which we have no profile) are also regarded as valid.

Suspicious access. In cases where the access is within a buffer that is not contained in the profile for the current program state, we report a suspicious access. While in the case of library functions that operate on buffers, such as `memcpy`, this might be a legitimate behavior, it could also be an overread from a neighboring buffer.

Out of bounds access. If the program accesses memory at a region outside any known buffer allocation sites, but the profile lists buffers for this program state, an out of bounds access is highly probable. When the referenced memory address is a symbolic expression, even stronger conclusions can sometimes be drawn: if the expression can evaluate to both an address within a buffer as well as outside it, an out of bounds access is possible.

3. GUIDANCE

Instead of trying to cover all the code of the program under test, we direct our efforts toward a specific, potentially vulnerable point in the program. For this, precise guidance of the symbolic execution is crucial: We need to avoid getting lost in uncontrolled state-space explosion. Instead, we want to quickly and comprehensively explore the execution paths that are most relevant to the vulnerability we are trying to trigger. This guidance takes several forms. First of all, we can use the control flow information collected in the preliminary analysis phase as a "map" that helps us to quickly lead execution towards the potentially vulnerable spot. Furthermore, we can take into account the path constraints associated with each state in the symbolic execution. Specifically, we are interested in any constraints on the bytes of input on which the address of the memory accesses we are testing may depend. As we will see, these constraints can help us reach the target program point along paths that are interesting because they differ in the way they calculate the address for the targeted memory access.

BORG starts by using concolic execution (Section 3.1). This provides us with an initial, successful execution of the program based on real world input data. From then on, several assessment functions evaluate each state based on criteria that are relevant for triggering the target vulnerability. Depending on the importance of each criterion, the combination of the assessment functions establishes a ranking of all the states. This ranking is computed periodically, and the top ranked state is then selected for execution.

In detail, the assessment involves the following steps:

1. Evaluate whether the target instruction is reachable from each state.
2. Based on the path constraints, evaluate whether an out-of-bounds access is possible from each state.
3. Rank states based on their proximity to the vulnerable spot.
4. At a branch point, evaluate whether a newly forked state would either exit or stay in a loop.
5. Among freshly forked states, prefer those whose path constraint affects parts of the input that are involved in the targeted buffer access.

Both reachability of the target instruction (1) and the satisfiability of the out-of-bounds constraint (2) are strong selectors: we are not going to explore states in which either the targeted access is unreachable, or the path constraints rule out a violation of the integrity assumption. Therefore, such states are simply discarded. All states that pass these filters are evaluated for further exploration. The following assessment criteria do not rule out states in general. Instead, they express a preference in which states will be explored first. The proximity metric (3) measures an approximation of the minimum number of instructions needed to reach the target instruction from each state. We discuss this metric in Section 3.2. To try to trigger an overread, we prefer states that stay inside a loop to states that exit the loop prematurely (4). Our handling of loops is discussed in Section 3.3. Finally, we try to explore states with a variety of path constraints involving the inputs that affect the targeted buffer access (5). Our use of path constraints for selecting states is discussed in Section 3.4.

3.1 Concolic Testing

Concolic testing [32] complements symbolic execution by backing the symbolic input data with a concrete value assignment. This concrete input data can be used to make a branch decision: the symbolic executor will, by default, follow the branch for which the path constraints evaluate to true based on the concrete assignment.

With BORG, we use concolic execution to get an initial, successful execution path through the program that allows to explore states deep in the program's logic. As input we use the real-world data from the preliminary testing phase that revealed the vulnerable spot.

Every path that is generated during symbolic execution requires a concrete assignment that match the constraint set of the path. While a new concrete assignment can be generated by the constraint solver, this will likely overwrite values from the original input. With BORG we strive to preserve as much concrete information from the original input as possible as it contains realistic data that allowed us to reach deep states. Therefore, whenever we pick a branch that leaves the initial execution path, we examine the path constraint generated by that branch and determine the exact parts of the input that caused this path constraint. We then replace only these parts of the original assignment with new values obtained by querying the constraint solver.

An important side-effect of always having a valid, concrete assignment of the input available for every state is that it will speed up symbolic execution significantly. During symbolic execution, the engine often needs a concrete sample value for a given symbolic expression, for example S2E uses this for its internal memory management. In such a case, the engine would normally issue a costly query to the constraint solver for a value. This is of special significance with memory operations: with plain symbolic execution, the constraint solver needs to be queried for a valid value whenever a memory operation depends on symbolic input and the address expression needs to be evaluated against the path constraints. If such a memory operation is executed within a loop, this can cause a significant slowdown. Provided that we always have a valid, concrete assignment at hand, however, the address expression can be evaluated in a straightforward manner.

3.2 Proximity Rating

To exercise the vulnerable spot as often as possible in different execution paths, we pick states that are in close proximity to the target instruction. The rationale behind this approach is that the closer we are to the target, the fewer branch instructions will be in our way that can lead us astray.

To estimate proximity, we require a distance metric between a program's current execution state and a target instruction. The general requirement for the metric is an ideal tradeoff between *fast* and *precise* computation that can be performed at every execution state change of the program. We define an execution state as the combination of the current basic block executed and the associated callstack. As a distance metric, BORG uses the minimum number of instructions that have to be executed to reach the target instruction from the current execution state.

To accurately compute this distance, we require precise information about the control flow of the program. For this, we rely on the control flow graph that was extracted during the preliminary analysis phase. However, standard graph search algorithms cannot be directly applied to searching within a control flow graph, as they do not take the call-return semantics into account: return edges must always match their preceding call edges. The main idea of our solution is to pre-calculate distance information for all execution states that can reach the target instruction by performing a backward search from the target instruction. Since it is not feasible to compute reachability and exact distance to the target for all possible program states in advance, we split the problem into two steps:

1. Static, offline calculation before guided testing. Function `calcdist` of Algorithm 1 calculates distance information based on the control flow graph.
2. Lightweight on-the-spot calculation during guided testing to calculate the actual distance to the target for a given program state based on the distance information.

Generating Distance Information.

To address the fact that storing one distance per execution state does not scale to large programs and that programs often call a function more than once, we devised an efficient way of storing distance information. We split the information into *absolute* and *relative* distance information. Relative distances are measured intraprocedurally, for a given basic block they are defined as the minimum distance until the function returns. Absolute distances are measured inter-procedurally, for a given basic block they are defined as the minimum distance until the target instruction is reached. To generate these different kinds of distance information, the algorithm operates in two modes: If the `rel` parameter is set, intraprocedural distances are calculated, otherwise interprocedural distances are calculated.

The `calcdist` function follows a typical worklist approach that will start from the target instruction and traverse the CFG backwards. As input it requires the predecessors (`pred`) for each basic block and knowledge about whether a basic block is a callsite (`cs`) or ends a function (`ret`). In addition, it requires a mapping between function exit blocks and their corresponding function heads (`fh`) as well as return sites and their call site (`call`). All this information is retrieved from the interprocedural CFG.

Algorithm 1 Offline distance information generation.

```
Require: pred[bb] ∀ bb ∈ basic_blocks
Require: ret[bb] := true iff bb ends with return
Require: cs[bb] := true iff bb is a callsite
Require: call[bb] callsite for a given return site
Require: fh[bb] function head for a given function exit
 1: function calcdist(target:address,rel:bool)
 2:     worklist ← {target}
 3:     if rel then
 4:         dist ← rel_dist
 5:     else
 6:         dist ← abs_dist
 7:     dist[target] ← 0
 8:     for c ∈ worklist do
 9:         worklist ← worklist \ {c}
10:         for p ∈ pred[c] do
11:             if cs[p] ∧ rel then
12:                 continue
13:             if ret[p] then
14:                 calcdist(p, true)
15:                 d ← dist[c] + size[call[c]] + rel_dist[fh[p]]

16:                 p ← call[c]
17:             else
18:                 d ← dist[c] + size[pred]
19:             if d < dist[p] then
20:                 dist[p] = d
21:                 worklist ← worklist ∪ {p}
```

The operation of the function is based on the well-known Dijkstra algorithm [16], with basic blocks as nodes, predecessors being their neighbors and distance information initialized to infinity except for the target. As long as the function $F(c)$ of the current basic block c is equal to that of a predecessor p, i.e. $F(p) == F(c)$, the algorithm proceeds like ordinary Dijkstra. However, predecessors breaching a function boundary need special treatment:

Function return. If p is a return from $F(p)$ to $F(c)$ (line 13), we recursively invoke `calcdist` for $F(p)$. To calculate the minimum distance of executing $F(p)$, we need distances relative to p and thus run `calcdist` in intraprocedural mode with p as the target. Once $F(p)$ is explored and `calcdist` returns, we continue with the callsite of $F(p)$ in $F(c)$, i.e. the intraprocedural predecessor of c (line 16). To calculate the distance of the callsite, we sum up the distance of the current block $dist[c]$, the distance of the function head of $F(p)$, $rel\_dist[fh(p)]$ and the size of the basic block that contains the callsite $size[call[c]]$ (line 15). The remaining procedure is the same as with an ordinary predecessor: If the resulting distance is lower than a previously stored distance, we update its distance and add it to the worklist.

Function call. If p is a call from $F(p)$ to $F(c)$, processing depends on the mode the algorithm is operating in. In intraprocedural mode, we are restricted to the scope of $F(c)$ and thus skip p (line 12). In interprocedural mode, we treat p like a normal predecessor as we are calculating absolute distances.

Putting it together, `calcdist` is invoked in interprocedural mode with the given target. Whenever it hits a predecessor that is returning from a function, it switches to intraprocedural mode to calculate relative distances within that function.

Since we only store the information per basic block and not per state, memory consumption scales linearly to the size of the program. Apart from that, we do not explore functions multiple times once they have been covered completely.

Calculating Actual Distances.

During guided testing, we use the distance information to calculate the actual distance for a given execution state. The execution state is given by a stack of return addresses with the address of the current basic block on top. The optimum distance to the target is then calculated by summing up the distance information of the basic blocks on the callstack. Beginning with the item on top of the stack, we stop at the first item for which absolute distance information is available. If an item's distance information is unknown (i.e. the target is unreachable), we set it to infinity. This on-the-spot calculation's runtime is linear in the number of items on the callstack and can thus be performed very fast.

It is possible that during the guided testing phase, execution reaches a basic block that is not part of the CFG constructed in the preliminary analysis phase because of incomplete indirect call target information. Such basic blocks need special treatment, because we do not have distance information on them. For states that have reached new code, we retain the state's last distance assessment and dynamically enhance the CFG with the new information. As soon as we are back on a known basic block, we pause the guided testing and re-run Algorithm 1 on the expanded CFG.

3.3 Loop Handling

Branch points within loops can lead to a massive state explosion in symbolic execution. Thus a common guiding approach for symbolic execution is to perform one loop iteration and exit as soon as possible. While this is a reasonable strategy if just aiming for code coverage, it is less suited for our class of bugs. Programs typically use loops to parse and validate their inputs. Among the case studies in this work for instance, lighttpd and SSSD use loops for input validation, while libexif iterates over the input to return the index of a specific marker. In all cases triggering the bug depends on a successful, exhaustive completion of the loop.

Figure 3 shows the control flow of such an input processing loop that iterates over every character of the input string. Branch points B1-B3 will exit the loop prematurely on a special control character, while branch point B4 will iterate through the loop until it encounters a terminator. Given a target some place after the loop exit, proximity guidance will always pick one of the break edges as their distance to the target is smaller than completing the iteration or even taking the back edge. This way we would, per input character, explore three unsuccessful states before going for another iteration.

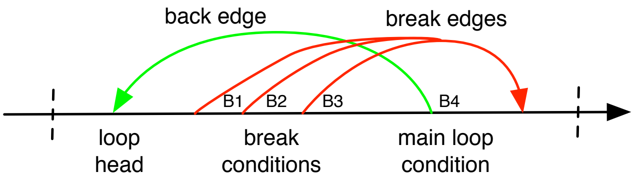

Figure 3: A typical input processing loop. States forked at branch points B1-B3 will be preferred by the proximity rating over taking the back edge.

Our solution to this issue is to prefer branch targets that stay within the loop. To pick branch targets that stay within the loop, we perform an intra-procedural loop detection for every branch point: We apply the Kosaraju-Sharir [34] algorithm on the intra-procedural control flow graph to identify strongly connected components. If the branch point is actually within such a component (i.e. loop), we examine its branch targets. Only if one of them points outside and one inside the loop, we mark the outside target as a loop exit.

3.4 Security Constraints

The goal of BORG is to find an execution path that performs an out-of-bounds access, thus violating an integrity assumption. Since we are targeting tainted accesses, the memory address accessed is a symbolic expression that has a relationship with some parts of the input. A typical scenario would be a pointer dereference where the program calculates the final memory address m by adding a symbolic offset to a concrete base address. Combining this information with the supposed target of the access, we can create security constraints that specify an out-of-bounds condition. In the case of a buffer that ranges from a start address s to an end address e, the security constraint SC would read $s \leqslant m \wedge m < e$. By querying the solver with the combination of the normal path constraints and this security constraint, we can filter any state as soon as $\neg\mathsf{SAT}(PC \wedge \neg SC)$.

We can also leverage another feature of security constraints for guidance. During symbolic execution we do not just see the final address m in case of a symbolic memory access, but the whole expression that represents all the calculation steps that involved symbolic input. Knowing which input bytes are involved in a security constraint allows us to prefer states that have been forked at branch points whose path constraints affected those very input bytes. This way we can explore the possible values of m more efficiently.

4. OVERREAD DETECTION

In contrast to overread detection that works at the source-code level like AddressSanitizer [33], BORG has to overcome the loss of high-level semantics at the binary level: Instead of an access expression like $buf[i]$, which readily provides the target buffer as well as the index of the read, BORG just sees a read access to a memory location.

Therefore, BORG first needs to associate memory accesses with buffers. On each memory access, it checks whether the accessed address falls inside the range of a buffer. To do so, it needs to identify the exact locations of buffers in memory beforehand. From a program's point of view, buffers can be located on the stack, on the heap or, in the case of global variables, in the bss or data segments.

While buffers on the heap, or global buffers in bss, or in the data segment are easy to track as they stay at the same memory location once they are allocated, stack-based buffers require more effort. Both their addressing as well as their scope depends on the current stack frame. They need to be validated and invalidated upon entering and exiting the function in which they have been declared. To calculate the actual memory location we need an offset of the stack buffer that is either relative to the base frame pointer or the stack pointer on function entry. While we resort to DWARF information for this special task only, the information can also be obtained for stripped binaries by statically identifying large variables located on the stack as suggested in [17] or by more sophisticated methods that evaluate dynamic access patterns [35]. For heap-based buffers we intercept all common allocation routines, such as `malloc` and `free`. Again, we focus on general purpose allocation routines like `malloc()` and `mmap()` in our implementation. In case of custom memory allocators, we can use tools like MemBrush [11] to extract these routines first.

To be able to check accesses, we need a way to refer to buffers across multiple executions of the program under test. Global buffers can simply be identified by their absolute address. For stack buffers, we record the allocation site (i.e. the function) as well as the offset from either the stack pointer or the base frame pointer. Finally, for heap buffers, we hash the program state (callstack + program counter) of the allocation site (i.e., where malloc was called).

Library functions that have well-known semantics and operate on buffers allow us to use performance optimizations. For example, `memcpy` contains a loop that copies four-byte chunks from source to destination. If an address provided to `memcpy` is symbolic, our system needs to perform a costly query to the constraint solver at each iteration. However, knowing the semantics of `memcpy`, we only need to intercept the function call and issue a single query to check whether the supplied parameters would allow an out-of-bounds access.

5. IMPLEMENTATION

The implementation of BORG's core, the guided testing, is based on S2E, a full-system selective symbolic execution engine [12]. S2E itself uses the Qemu emulator [3] and modifies its dynamic translation engine TCG to translate the binary guest code to LLVM instead of native host code. The generated LLVM code can then be run in the KLEE [8] symbolic virtual machine. KLEE uses STP [18] as its constraint solver. Figure 4 shows how BORG is based on S2E. The S2E base layer links KLEE and Qemu and expands KLEE's state concept to the operating system level. S2E also provides a plugin API that is used by BORG's main components.

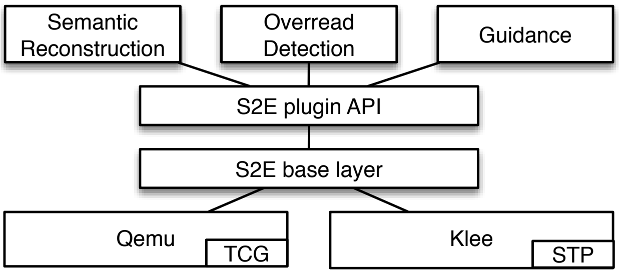

Figure 4: The BORG testing system. We implemented three extensions on top of the plugin API provided by S2E: Semantic view reconstruction restores OS concepts such as processes, overread detection monitors the process under test for buffer overreads and guidance tells S2E which states to execute.

The advantages of bringing symbolic execution to the operating system level are evident: binary applications can be symbolically executed in their native, real environment, even if they depend on libraries and devices. On the other hand, this high degree of symbolic execution support and integration at a comparably low level comes at a price. For instance, when symbolic input arrives via a network socket, it has to pass through all of the network stack in the kernel, adding significant overhead in symbolic execution. Similarly, even though S2E employs a copy-on-write memory concept, it still has to keep track of the whole OS state on a fork. Performance aside, a problem with operating at the level below the operating system itself is the lack of semantic information at the system level–an issue that we discuss in more detail presently.

Semantic view reconstruction. To improve execution performance of unrelated code, S2E needs to know when the program under test is executed and where it resides in memory. We implemented a plugin for S2E to reconstruct this information for Linux. Whenever the kernel starts a thread, the plugin extracts the memory layout and the name of the new task from the guest's memory. It then associates this information with the task's page table base address, which is unique per process and thus well-suited as a process identifier. We further track whether the program under test has terminated and we can thus kill the corresponding state. For the special case that the program is terminated by a signal, we additionally intercept signals to be able to report e.g. segmentation violations.

Tracking the program state. In BORG, all buffer accesses are associated with a program state. As already mentioned in Section 2.1, we define the program state as a combination of the address of the current instruction and the associated callstack. To remove all possible ambiguity, the callstack is represented by the return addresses, as they identify not only the function invoked, but also the callsite.

While determining the current instruction pointer is straightforward, the callstack needs more effort. In a nutshell, we keep track of all function invocations and returns throughout the program execution on a per-state basis. Since programs are not bound to follow the call-return semantics at the assembly level, we maintain the callstack on a best-effort basis.

The reasons for not adhering to the normal call-return convention vary. The best known example is position independent code. Such code may need to determine its current address; A popular way to do so is by means of a call instruction, as it pushes the address of the next instruction onto the stack. Besides, calls without returns also occur in exception handlers and event-based programming. To account for these cases, our callstack handling keeps track of the expected return addresses. Upon executing a return instruction, we distinguish two cases: If a return target is not contained in the callstack, the callstack is left untouched. Otherwise, we pop all items from the callstack until we reach the matching return address.

Preliminary analysis. For the static phase of the preliminary analysis, we leverage the IDA Pro disassembler. Our CFG generation script starts with the main function and descends into every called function, splitting the code on control transfer boundaries into basic blocks. To compensate for the shortcomings of this static approach, we additionally explore all functions we encounter during dynamic analysis and also extract the targets of indirect jumps.

For the dynamic phase we first run the program under test in a modified version of the lightweight taint tracker Minemu [4]. This will output all memory accesses encountered that depend on tainted input data. To produce the buffer profiles, we additionally run the tested program in S2E with symbolic execution completely disabled and only the semantic reconstruction and overread detection plugins enabled.

6. EVALUATION

To evaluate the effectiveness of BORG, we selected a number of recent out-of-bound vulnerabilities in real-world server applications and commonly used libraries. We selected the programs by querying the NVD database of NIST [28] for the most recent and critical buffer overread vulnerabilities in open source programs that S2E was able to execute. We used open source programs and known bugs so we could verify the results accurately.

Table 1 lists characteristics of the programs under test as well as results from being analyzed by BORG. The *basic block count* is an indicator of both size and complexity of the program. It shows that BORG also works on large applications such as ClamAV with over 38,000 basic blocks. The *candidate* column lists how many critical accesses were identified by BORG's target selection. For libexif our target selection could not come up with candidates, so we chose to use the selection strategy of Dowser [23] for this case. The size of the *symbolic input* supplied to each program under test varied significantly, based on the kind of input processed by the program: While the communication protocol of SSSD uses only a small 42 byte message, the test input to libmagic, an MS Office document, was more than 170 times as large. The sum of the *preliminary analysis* and *guided testing* column states how long it took BORG to trigger a vulnerability. Finally the *state space* is the number of states generated during symbolic execution and the *states explored* the part of the state space that was explored by guided testing.

In the following we will provide insight on the programs and on how BORG dealt with their vulnerabilities.

Lighttpd. Lighttpd is a lightweight, yet standards-compliant web server that is used, amongst others, by YouTube. Using the supplied test cases for authentication, BORG found two tainted memory accesses during preliminary analysis. One of them corresponds to a buffer overread vulnerability specified by CVE-2011-4362. It is located in a function that handles base64 decoding of the username supplied in an HTTP request with basic authentication, where part of the input is used as an offset into a conversion table:

```
 1 static unsigned char * base64_decode(buffer *out,
       const char *in) {
 2   unsigned char *result = (unsigned char *)out->ptr;
 3   int ch = in[0];
 4   for (size_t i = 0; i < strlen(in); i++) {
 5     ch = in[i];
 6     if (ch == '\0' || ch == base64_pad) break;
 7     ch = base64_reverse_table[ch];
 8     if (ch < 0) continue;
 9     switch(i % 4) {
10       case 0:
11         result[j] = ch << 2;
12         break;
13                              ⋮
14       }
15     }
16   return result;
17 }
18 int http_auth_basic_check(..., const char *realm_str)
       {
19   buffer *username, *password;
20                              ⋮
21   username = buffer_init();
22   if (!base64_decode(username, realm_str)) return 0;
23                              ⋮
24 }
```

The realm_str pointer (line 18, 22) directly references input data. The base64_decode function will iterate over each byte of the input (line 4) unless it encounters a termination character. It decodes each character by means of the base64_reverse_table (line 7) and writes the result to the output pointer. The problem is the missing bounds check for ch when using it as an offset into the array in line 7. Since the array has 256 entries one might first not think that an overread is possible, as this perfectly matches the maximum value of an unsigned character. However, before being used as an index, the character is first casted to a signed integer (line 3). Therefore all values above 0x7f will turn into negative offsets into the table.

Still, the generated out-of-bounds constraint does not evaluate to true in the initial state, as the path constraints do not allow for it. After guidance dismissed all irrelevant states using the security constraints, the closest valid states based on the proximity ranking pointed to a validation loop that happens at a very early stage in the program. As it turns out, each character of the request header (line 2) is first checked for validity:

```
1 for (; i < con->parse_request->used && !done; i++) {
2   char *cur = con->parse_request->ptr + i;
3                              ⋮
4   if (*cur >= 0 && *cur < 32 && *cur != '\t') break;
```

At this point we fork a state for every header character. The compiled assembly version of this if-clause will only pass if *cur is negative, bigger than 0x1f or if it is not equal to 0x9. Our test input passes this if-clause, because all characters' values exceed 0x1f. The path constraints generated corresponding to this input are the

Program	Basic Blocks	Candidates	Symbolic Input	Preliminary Analysis		Guided Testing	State Space	States Explored
				static	dynamic			
Lighttpd	7,139	2	63 byte	3s	1m 52s	3m 36s	369	10
FFmpeg	4,654	4	5,120 byte	2s	10m 42s	3m 54s	10	1
libmagic	512	14	7,168 byte	<1s	19s	3h 46m 37s	762	1
libexif	2,506	(87)	678 byte	<1s	10s	1m 45s	1,804	503
SSSD	9,938	4	42 byte	6s	2m 40s	2m 56s	116	7
ClamAV	38,037	12	3,505 byte	51s	5m 26s	12h 40m 09s	3,792	1

Table 1: Programs analyzed using BORG. For libexif, we used Dowser's selection strategy.

reason why an invalid offset was not possible for the initial path: they effectively limit the possible range for values to 0x1f-0x7f. By taking a different branch that allows for a different set of input characters, BORG can trigger the vulnerability. In this case loop handling prefers states that do not lead to the break condition.

Given the depth of the bug and the fact that BORG forks hundreds of states until it triggers the vulnerability, an uninformed exploration strategy is highly likely to fail – each path that does not hit or trigger the bug will fork off a large number of new states during exploration.

SSSD. The System Security Services Daemon (SSSD) is installed per default on Fedora-related Linux distributions. It provides remote access to various identity and authentication resources through a common framework. We tested the SSH-module, which is susceptible to a buffer overread specified in CVE-2013-0220. As the supplied test suite is only targeted at testing the internal database, we intercepted network traffic from the supplied client. Based on this test input, BORG identified four target candidates, all of them located in a function that parses the incoming request.

```
 1 static errno_t ssh_cmd_parse_request(struct
       ssh_cmd_ctx *cmd_ctx) {
 2     struct cli_ctx *cctx = cmd_ctx->cctx;
 3     uint8_t *body;
 4     size_t body_len;
 5     size_t c = 0;
 6     uint32_t flags, name_len, alias_len;
 7     char *name;
 8
 9     sss_packet_get_body(cctx->creq->in, &body,
           &body_len);
10     memcpy_c(&flags, body+c, body_len, &c);
11     memcpy_c(&name_len, body+c, body_len, &c);
12     if (flags > 1) {return EINVAL;}
13     if (name_len == 0) {return EINVAL;}
14     name = (char *)(body+c);
15     if (!sss_utf8_check(name, name_len-1) ||
           name[name_len-1] != 0) {
16         return EINVAL;
17     }
18     c += name_len;
19                            ⋮
20     if (flags & 1) {
21         memcpy_c(&alias_len, body+c, body_len, &c);
22                            ⋮
```

The body pointer used in this function references part of the data received from the network. The memcpy_c function is a wrapper around the ordinary memcpy that will add the length parameter to its last parameter. The problem is the name_len variable that is parsed from the input data in line 11 and, without any further validation, used as an offset in lines 15, 18 and 21. However, the out-of-bounds constraint does not evaluate to true on the initial path, because the sss_utf8_check function in line 15 generates path constraints that limit name_len to the actual length of the corresponding part of the input. Proximity rating and security constraints take us back to the states that have been forked last in the validation loop in sss_utf8_check. Loop handling dismisses numerous states that would exit the loop prematurely. One of the high-ranked states triggers a different behavior of sss_utf8_check: if the length parameter is negative, it assumes a zero terminated string and returns true if the characters up to the termination character are in conformance to UTF8. Since this is the case, name_len is now negative and all subsequent accesses that use it as part of their offset potentially overread. This case shows that BORG is also effective using observed real-world input data when no test cases are available.

FFmpeg. FFmpeg is a collection of libraries that provide functionality to record, convert and stream audio and video of various formats. Since it is not a standalone program, we implemented a very basic test program that will open, identify and read a file. As concrete test input we used a typical stream in the DV format, truncated to a limited number of frames. Among the four candidates given by BORG's target selection was the overread vulnerability specified by CVE-2011-3936.

```
 1 static const uint8_t *dv_extract_pack(uint8_t *frame,
       enum dv_pack_type t) {
 2   int offs;
 3   switch (t) {
 4   case dv_audio_source:
 5     offs = (80*6 + 80*16*3 + 3);
 6     break;
 7                            ⋮
 8   }
 9   return frame[offs] == t ? &frame[offs] : NULL;
10 }
11 static int dv_extract_audio(uint8_t *frame, ...) {
12                            ⋮
13   as_pack = dv_extract_pack(frame, dv_audio_source);
14   if (!as_pack) return 0;
15   freq  = (as_pack[4] >> 3) & 0x07;
16   size = (sys->audio_min_samples[freq] + smpls) * 4;
17 }
```

In this case, part of the input is modified by a number of logic operations before it is used as an array offset: The frame pointer passed to dv_extract_audio is a direct reference to the supplied input. Since our input has the dv_audio_source type set at the offset required by dv_extract_pack, symbolic execution follows the else branch of the conditional and continues. Now freq is calculated by masking a value of as_pack (that refers directly to the input) with 0x7. In line 16, freq is used as an offset into the audio_min_samples array that only contains three entries, although the bitmask allows values up to seven. Since no (path) constraints apply to the value in as_pack[4], the out-of-bounds constraint we generate is already satisfiable and BORG reports the possible overread and outputs a test case without exploring additional states.

Libmagic. Libmagic is the library behind the popular "file" command line tool that will try to determine the type of a given file based on its content. It is also used by larger programs such as the Apache web server. While most file types are covered by magic bytes contained in the magic database, libmagic performs further processing for a limited number of file types, such as CDF, Microsoft Office's composite document format. Since libmagic does not come with a testfile for the CDF format, we used Microsoft Office to produce an empty CDF document. BORG's target selection identified 14 candidates, one of them being an overread specified by CVE-2012-1571. In this case, the overread is caused by lacking validation of the source pointer on a call to memcpy. Again, BORG is able to query the constraint solver for a test case that triggers the vulnerability in the first state, since the path constraints allow for it.

The comparably long time it took until the vulnerability was triggered is caused by the combination of a rather large symbolic input size and the large number of constraints generated. This causes queries to the constraint solver to take up to 90s each.

Libexif. Libexif reads and writes EXIF metadata information from and to JPG image files. It is a popular library used by various image processing applications such as ImageMagick. Being a library, we tested its main functionality that extracts EXIF data from a memory region. As input we used a sample JPG image shipped with the libexif library.

In this case, BORG's target selection could not come up with candidates because none of the buffer accesses used tainted input for the address calculation. Having the source code available, we chose to try Dowser's selection heuristic in this case. While being based on taint tracking as well, it employs special methods to handle implicit flows. Indeed, it provided us with an access affected by CVE-2012-2836 that is ranked at place 13 out of 87 loops. In the following BORG's guided testing managed to trigger the vulnerability, showing the effectiveness of the proximity rating.

Compared to the other examples, the bug in libexif is located at a rather shallow position in the program's logic. Still, when we turned off our guiding enhancements and ran libexif with concolic execution and a standard depth-first exploration guidance, the bug was not triggered within 29 hours.

ClamAV. ClamAV is a widely used anti-virus engine that offers its capabilities in a library. When used on a password-protected PDF file, target selection identified 12 security critical accesses. This is in accordance with CVE-2013-2021, which indicates an overread vulnerability with encrypted PDF files: The user password checking function in ClamAV's PDF scanner passes user input directly to memcpy as a length parameter. Since the out-of-bound constraints already hold in the first state, BORG was able to generate a test case for the overread right away. As ClamAV is by far the biggest program we applied BORG on, its size is being reflected by the long time it took to trigger the bug compared to the other programs.

7. RELATED WORK

The use of symbolic execution to find bugs in programs has been subject to a substantial body of recent research, including CUTE [32], DART [19], EXE [9], KLEE [8], SAGE [20]. Other than SAGE, these approaches require source code. Unlike BORG, their goal is to achieve high code coverage of the tested program and find bugs throughout the program.

SE targeted at specific bugs. A number of papers investigate how symbolic execution can be applied in a more targeted fashion to attempt to detect specific bugs.

In [23] we present Dowser, a guided symbolic execution strategy geared towards buffer overflow bugs. It uses static analysis to identify possible locations for buffer overflow bugs within loops and ranks them based on a metric that evaluates the complexity of the access. Dowser's static analysis techniques heavily rely on the expressiveness of source code, porting them to support binaries represents a challenge in itself that we leave to future work. While Dowser's heuristics for target selection can enhance BORG's efficiency, it is not sufficient by itself, as it only focuses on accesses within loops. It thus fails in all our evaluation examples except for the libexif case.

Cui et al. [15] aim to discover bugs in the enforcement of higher level policies, such as heap or file descriptor misuse. They use path slicing to statically infer execution paths that are not relevant for the given testing run. While the paper demonstrates the effectiveness on coarse-grained bugs, program slicing is less suitable for dealing with fine-grained bugs like memory errors: In a typical application that first parses the input before the program logic takes effect, the parser will be data-dependent on the whole input, thus leaving nothing for path slicing to prune. The same applies to DiSE [29] and its simplification based on "unaffected" nodes. Program slicing is also significantly more difficult to perform on stripped binaries where pointer aliasing is an unsolvable problem.

In Zesti [27] the authors use symbolic execution to improve testing of sensitive instructions. They limit their exploration to paths that do not exceed a certain fork depth, measured backwards from a targeted sensitive instruction. In order to scale, they use a fork depth of ten forks. Thus their approach would fail with programs such as lighttpd where the path constraint that allows for a bug to occur is in a validation loop "far" (in terms of fork depth) from the actual bug.

Babić et al. [2] describe a guided analysis method similar to BORG that, in contrast to the aforementioned systems, also works on binaries. The authors use a data-flow graph to identify possible vulnerabilities first, and a shortest-path distance heuristic based on a control flow graph as a second step to guide symbolic execution to the vulnerabilities. In contrast to this work, BORG's selection strategy also uses dynamic analysis to both resolve indirect call targets and dynamically allocated memory. BORG additionally uses security constraints for guidance and employs a more fine grained distance metric that uses the current basic block and the runtime call stack, while this system only uses the current basic block. Finally, unlike BORG, this work has not been tested on real-world programs, but only historic and simplified vulnerabilities.

SE aimed at known vulnerabilities. Symbolic execution has also been leveraged to replay or extend existing vulnerabilities. Such approaches typically involve strong guidance strategies, but are based on stronger assumptions regarding existing information about faulty application behavior. ESD by Zamfir et al. [39] applies a distance based guidance strategy aiming to replicate inputs responsible for crashes based on existing core dumps. Even though their distance heuristic is rather coarse, their method proves that distance based guidance is a strong tool for eliminating the potential exponential explosion of symbolic execution. On a similar note, some researchers aim to increase the threat level of existing bugs using guided symbolic execution. Avgerinos et. al. [1] and Cha et al. [10] developed techniques for guided symbolic execution with the purpose of exploit generation. They assume knowledge about existing bugs that are considered benign from a security perspective and transform them into full blown security exploits.

Approaches to guidance. The general idea of using the program's structure in form of a CFG for guiding symbolic execution was introduced by [7]. Ma et al. [26] show that distance calculation is generally suited to navigate symbolic execution towards specific target instructions, but might lead to inferior results when a longer path is required. While they propose a form of backward symbolic execution that is not feasible with an online symbolic executor, BORG addresses these problems by its additional guiding techniques. Both Rungta and Cho [13, 30] leverage program models for guidance.

While models are suited to guide symbolic execution to specific high-level states, they are only a coarse approximation of the real program that is possibly incomplete and might thus miss specific bugs. Fitnex [38] selects those paths that are more likely to satisfy a certain target predicate. BORG does not use fitness guidance because a measure of how well an access is within bounds does not relate to how likely the access might go out-of-bounds. In contrast, BORG's boolean security constraints prune states that are in bound and put much less strain on the constraint solver.

Improving symbolic execution. A significant body of research focusing on improving symbolic execution in general. Approaches include parallelization [5], swapping states to disk [10], efficient handling of array accesses in loops [31], summarizing loops [21] and state merging [6, 25]. By extending the underlying symbolic execution engine with the techniques presented above, the overall performance of BORG would improve, enabling even larger applications to be tested. The benefit of guidance would still be relevant since it is offers a relative improvement to the underlying symbolic execution engine.

8. EXTENDING BORG

Our overall approach and the guidance techniques discussed in Section 3 have applications beyond the ones we address in this work, which only targets a specific class of bugs (buffer overreads), and the subset of bugs in this class that can be targeted by the selection strategy we propose.

To be able to effectively apply our guided symbolic execution approach to a class of bugs, we need to have at least one target selection strategy that is able to identify a limited number of program instructions that could potentially trigger a bug of that class. Note that, to be useful, a target selection strategy does not need to be able to target all possible bugs within a class: It is enough that it can target some bugs that would be hard to find with unguided testing. A comprehensive, practical approach to automated bug detection could then be to execute both unguided testing runs that aim to maximize code coverage, and guided testing runs led by a number of selection strategies for different types of bugs. In the following, we will discuss a few possible selection strategies for different classes of bugs.

Buffer Boundary Violations. All of the techniques that BORG applies to buffer overreads can trivially be adapted to similarly detect buffer overflows. As discussed, our target selection strategy for this application targets memory accesses that use tainted memory address. To detect additional overread and overwrite bugs, we could introduce additional selection strategies, such as the one proposed in Dowser [22, 23].

Format String Vulnerabilities. Comparable to buffer boundary violations, format string vulnerabilities can also be exploited to write to or read from a program's memory in unexpected and potentially harmful ways. These vulnerabilities occur when attackers are able to control the format string provided as input to functions in the *printf* family and to introduce unexpected modifiers, such as %x or %n. Format string modifiers can cause the function to read data from and write data to arbitrary memory regions.

For this class of bug, an effective selection strategy is to target all invocations of functions in the *printf* family where the format string is derived from user input. Often logging functions turn out to be such candidates. During exploration, guidance can automatically assess whether the path constraints allow for certain modifiers to end up in format strings at all. Finally, whenever a candidate function is executed, the detection component can check whether it will actually access memory regions it is not supposed to.

Control Flow Integrity. If an offset into a jump table is calculated based on program input, malformed input that is not sufficiently validated may cause a violation of control flow integrity: The program could either pick a target in the table it is not supposed to jump to at this moment, or, if the final address is not bounds-checked against the size of the table, it could jump to an arbitrary memory location.

One selection strategy for this class of bugs can be to target all indirect control flow transfer instructions whose target depends on user input. During symbolic execution, guidance prefers states whose path constraints are relaxed enough to allow for a wide range of possible offsets. In the preliminary analysis phase we construct a profile of valid jump targets for every control flow transfer. Whenever a control flow transfer happens, the detection component can then check whether we can pick a target outside the ones recorded.

9. LIMITATIONS AND FUTURE WORK

BORG's effectiveness or applicability is limited by the target selection method. While the evaluation shows that the current selection method allows it to find deep bugs in complex, real-world programs, there is certainly room for improvement. One issue is that BORG's current selection strategy does not track implicit flows, which prevented it from finding candidates in the libexif case, as well as CVE-2012-1798 of libmagick and CVE-2012-3425 of libpng. We plan to look into possible solutions that have been proposed for this issue [24]. Another issue is BORG's dependence on concrete input: it cannot find bugs in code that is not covered by any test input. This limitation can be alleviated by introducing additional selection strategies that are based purely on static analysis.

Our overall approach and the guidance techniques discussed in Section 3 have applications beyond the ones we address in this work. In future work, we plan to develop a number of additional selection strategies as outlined in Section 8.

Finally, our implementation of symbolic execution, which is based on S2E, is rather slow and uses a large amount of memory per state. This prevented us from evaluating BORG on CVE-2012-2141 of net-snmp. A more efficient implementation of the core engine could allow BORG to find more bugs faster.

10. CONCLUSION

In this paper we presented BORG, a tool for finding out-of-bounds buffer accesses in binaries. Unlike many testing techniques, BORG does not aim to cover as much code as possible in the tested program. Instead, it selects specific points in the program that may be vulnerable, and then proceeds to thoroughly test them by exploring program paths that lead to those points, while trying to trigger and detect a bug. For this, we introduce techniques to guide symbolic execution and make it focus on execution paths that are most relevant to the selected targets and the bugs that may lurk there. As a result, BORG can output concrete program inputs that trigger a buggy behavior. Our novel guidance approach allows BORG to reach and trigger bugs deep in the logic of the programs under test as we show in case studies on six real-world applications.

While BORG focuses on detecting overread bugs, the guidance techniques we introduce could be used to target different classes of bugs, such as buffer overflows, format strings vulnerabilities and control flow integrity violations.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement 257007 (SysSec), the European Research Council through project ERC-2010-StG 259108 (ROSETTA), the MSR Ph.D. Scholarship 2011-049 and the FFG – Austrian Research Promotion under grant COMET K1.

11. REFERENCES

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[2] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the Symposium on Software Testing and Analysis (ISTA)*, 2011.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2005.

[4] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[5] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the European conference on Computer systems (EuroSys)*, 2011.

[6] S. Bugrara and D. Engler. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[7] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Washington, DC, USA, 2012.

[11] X. Chen, A. Slowinska, and H. Bos. Who allocated my memory? Detecting custom memory allocators in C binaries. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Koblenz, Germany, 2013.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[13] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2011.

[14] Codenomicon. The Heartbleed Bug. `heartbleed.com`.

[15] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

[17] C. Eagle. *The IDA Pro Book*. William Polloc, 2011.

[18] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2007.

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[20] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Conference (NDSS)*, 2008.

[21] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the Symposium on Software Testing and Analysis (ISTA)*, 2011.

[22] I. Haller, A. Slowinska, and H. Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the European Workshop on System Security (Eurosec)*, 2013.

[23] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the USENIX Security Symposium (USENIX SEC)*, 2013.

[24] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[26] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the Conference on Static Analysis*, 2011.

[27] P. D. Marinescu and C. Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[28] NIST. National Vulnerability Database. `web.nvd.nist.gov`.

[29] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[30] N. Rungta, E. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Model Checking Software*, LNCS. 2009.

[31] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Symposium on Software Testing and Analysis (ISTA)*, 2009.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference*, 2005.

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[34] M. Sharir. A strong connectivity algorithm and its applications to data flow analysis. In *Computers and Mathematics with Applications*, 1981.

[35] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[36] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[37] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections: Setting back browser security by 10 years. In *Blackhat*, 2008.

[38] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*, 2009.

[39] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the European conference on Computer systems (EuroSys)*, 2010.