2012 IEEE Symposium on Security and Privacy

Unleashing MAYHEM on Binary Code

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley
Carnegie Mellon University
Pittsburgh, PA
{*sangkilc, thanassis, alexandre.rebert, dbrumley*}*@cmu.edu*

***Abstract*—In this paper we present MAYHEM, a new system for automatically finding exploitable bugs in binary (i.e., executable) programs. Every bug reported by MAYHEM is accompanied by a working shell-spawning exploit. The working exploits ensure soundness and that each bug report is security-critical and actionable. MAYHEM works on raw binary code without debugging information. To make exploit generation possible at the binary-level, MAYHEM addresses two major technical challenges: actively managing execution paths without exhausting memory, and reasoning about *symbolic memory indices*, where a load or a store address depends on user input. To this end, we propose two novel techniques: 1) hybrid symbolic execution for combining online and offline (concolic) execution to maximize the benefits of both techniques, and 2) index-based memory modeling, a technique that allows MAYHEM to efficiently reason about symbolic memory at the binary level. We used MAYHEM to find and demonstrate 29 exploitable vulnerabilities in both Linux and Windows programs, 2 of which were previously undocumented.**

***Keywords*-hybrid execution, symbolic memory, index-based memory modeling, exploit generation**

I. INTRODUCTION

Bugs are plentiful. For example, the Ubuntu Linux bug management database currently lists over 90,000 open bugs [17]. However, bugs that can be exploited by attackers are typically the most serious, and should be patched first. Thus, a central question is not whether a program has bugs, but which bugs are exploitable.

In this paper we present MAYHEM, a sound system for automatically finding exploitable bugs in binary (i.e., executable) programs. MAYHEM produces a working control-hijack exploit for each bug it reports, thus guaranteeing each bug report is actionable and security-critical. By working with binary code MAYHEM enables even those without source code access to check the (in)security of their software.

MAYHEM detects and generates exploits based on the basic principles introduced in our previous work on AEG [2]. At a high-level, MAYHEM finds exploitable paths by augmenting symbolic execution [16] with additional constraints at potentially vulnerable program points. The constraints include details such as whether an instruction pointer can be redirected, whether we can position attack code in memory, and ultimately, whether we can execute attacker's code. If the resulting formula is satisfiable, then an exploit is possible.

A main challenge in exploit generation is exploring enough of the state space of an application to find exploitable paths. In order to tackle this problem, MAYHEM's design is based on four main principles: 1) the system should be able to make forward progress for arbitrarily long times—ideally run "forever"—without exceeding the given resources (especially memory), 2) in order to maximize performance, the system should not repeat work, 3) the system should not throw away any work—previous analysis results of the system should be reusable on subsequent runs, and 4) the system should be able to reason about symbolic memory where a load or store address depends on user input. Handling memory addresses is essential to exploit real-world bugs. Principle #1 is necessary for running complex applications, since most non-trivial programs will contain a potentially infinite number of paths to explore.

Current approaches to symbolic execution, e.g., CUTE [26], BitBlaze [5], KLEE [9], SAGE [13], McVeto [27], AEG [2], S2E [28], and others [3], [21], do not satisfy all the above design points. Conceptually, current executors can be divided into two main categories: offline executors — which concretely run a single execution path and then symbolically execute it (also known as trace-based or *concolic* executors, e.g., SAGE), and online executors — which try to execute all possible paths in a single run of the system (e.g., S2E). Neither online nor offline executors satisfy principles #1-#3. In addition, most symbolic execution engines do not reason about symbolic memory, thus do not meet principle #4.

Offline symbolic executors [5], [13] reason about a single execution path at a time. Principle #1 is satisfied by iteratively picking new paths to explore. Further, every run of the system is independent from the others and thus results of previous runs can be immediately reused, satisfying principle #3. However, offline does not satisfy principle #2. Every run of the system needs to restart execution of the program from the very beginning. Conceptually, the same instructions need to be executed repeatedly for every execution trace. Our experimental results show that this re-execution can be very expensive (see §VIII).

Online symbolic execution [9], [28] forks at each branch point. Previous instructions are never re-executed, but the continued forking puts a strain on memory, slowing down the execution engine as the number of branches increase. The result is no forward progress and thus principles #1 and #3 are not met. Some online executors such as KLEE stop forking to avoid being slowed down by their memory

DOI 10.1109/SP.2012.31

use. Such executors satisfy principle #1 but not principle #3 (interesting paths are potentially eliminated).

MAYHEM combines the best of both worlds by introducing *hybrid symbolic execution*, where execution alternates between online and offline symbolic execution runs. Hybrid execution acts like a memory manager in an OS, except that it is designed to *efficiently* swap out symbolic execution engines. When memory is under pressure, the hybrid engine picks a running executor, and saves the current execution state, and path formula. The thread is restored by restoring the formula, concretely running the program up to the previous execution state, and then continuing. Caching the path formulas prevents the symbolic re-execution of instructions, which is the bottleneck in offline, while managing memory more efficiently than online execution.

MAYHEM also proposes techniques for efficiently reasoning about symbolic memory. A symbolic memory access occurs when a load or store address depends on input. Symbolic pointers are very common at the binary level, and being able to reason about them is necessary to generate control-hijack exploits. In fact, our experiments show that 40% of the generated exploits would have been impossible due to concretization constraints (§VIII). To overcome this problem, MAYHEM employs an index-based memory model (§V) to avoid constraining the index whenever possible.

Results are encouraging. While there is ample room for new research, MAYHEM currently generates exploits for several security vulnerabilities: buffer overflows, function pointer overwrites, and format string vulnerabilities for 29 different programs. MAYHEM also demonstrates 2-10× speedup over offline symbolic execution without having the memory constraints of online symbolic execution.

Overall, MAYHEM makes the following contributions:

1) Hybrid execution. We introduce a new scheme for symbolic execution—which we call *hybrid* symbolic execution—that allows us to find a better balance between speed and memory requirements. Hybrid execution enables MAYHEM to explore multiple paths faster than existing approaches (see §IV).

2) Index-based memory modeling. We propose index-based memory model as a practical approach to dealing with symbolic indices at the binary-level. (see §V).

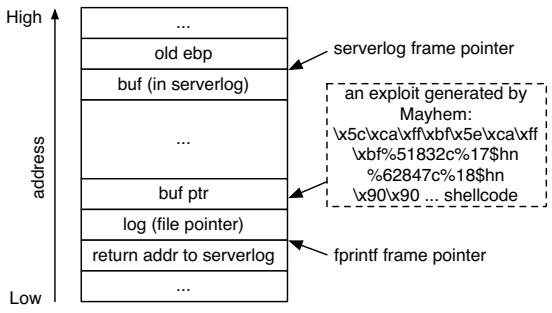
3) Binary-only exploit generation. We present the first end-to-end binary-only exploitable bug finding system that demonstrates exploitability by outputting working control hijack exploits.

II. OVERVIEW OF MAYHEM

In this section we describe the overall architecture, usage scenario, and challenges for finding exploitable bugs. We use an HTTP server, `orzHttpd` [1]—shown in Figure 1a—as an example to highlight the main challenges and present how MAYHEM works. Note that we show source for clarity and simplicity; MAYHEM runs on binary code.

```
1 #define BUFSIZE 4096
2
3 typedef struct {
4   char buf[BUFSIZE];
5   int used;
6 } STATIC_BUFFER_t;
7
8 typedef struct conn {
9   STATIC_BUFFER_t read_buf;
10  ... // omitted
11 } CONN_t;
12
13 static void serverlog(LOG_TYPE_t type,
14                        const char *format, ...)
15 {
16   ... // omitted
17   if(format != NULL) {
18     va_start(ap, format);
19     vsprintf(buf, format, ap);
20     va_end(ap);
21   }
22   fprintf(log, buf); // vulnerable point
23   fflush(log);
24 }
25
26 HTTP_STATE_t http_read_request(CONN_t *conn)
27 {
28   ... // omitted
29   while(conn->read_buf.used < BUFSIZE) {
30     sz = static_buffer_read(conn, &conn->
          read_buf);
31     if(sz < 0) {
32       ...
33       conn->read_buf.used += sz;
34       if(memcmp(&conn->read_buf.buf[conn->
            read_buf.used] - 4, "\r\n\r\n", 4) ==
            0)
35       {
36         break;
37       }
38   }
39   if(conn->read_buf.used >= BUFSIZE) {
40     conn->status.st = HTTP_STATUS_400;
41     return HTTP_STATE_ERROR;
42   }
43   ...
44   serverlog(ERROR_LOG,
45             "%s\n",
46             conn->read_buf.buf);
47   ...
48 }
```

(a) Code snippet.



(b) Stack diagram of the vulnerable program.

Figure 1: `orzHttpd` vulnerability

In `orzHttpd`, each HTTP connection is passed to `http_read_request`. This routine in turn calls `static_buffer_read` as part of the loop on line 29 to get the user request string. The user input is placed into the 4096-byte buffer `conn->read_buf.buf` on line 30. Each read increments the variable `conn->read_buf.used` by the number of bytes read so far in order to prevent a buffer overflow. The read loop continues until `\r\n\r\n` is found, checked on line 34. If the user passes in more than 4096 bytes without an HTTP end-of-line character, the read loop aborts and the server returns a 400 error status message on line 41. Each non-error request gets logged via the `serverlog` function.

The vulnerability itself is in `serverlog`, which calls `fprintf` with a user specified format string (an HTTP request). Variadic functions such as `fprintf` use a format string specifier to determine how to walk the stack looking for arguments. An exploit for this vulnerability works by supplying format strings that cause `fprintf` to walk the stack to user-controlled data. The exploit then uses additional format specifiers to write to the desired location [22]. Figure 1b shows the stack layout of `orzHttpd` when the format string vulnerability is detected. There is a call to `fprintf` and the formatting argument is a string of user-controlled bytes.

We highlight several key points for finding exploitable bugs:

Low-level details matter: Determining exploitability requires that we reason about low-level details like return addresses and stack pointers. This is our motivation for focusing on binary-level techniques.

There are an enormous number of paths: In the example, there is a new path on every encounter of an `if` statement, which can lead to an exponential path explosion. Additionally, the number of paths in many portions of the code is related to the size of the input. For example, `memcmp` unfolds a loop, creating a new path for symbolic execution on each iteration. Longer inputs mean more conditions, more forks, and harder scalability challenges. Unfortunately most exploits are not short strings, e.g., in a buffer overflow typical exploits are hundreds or thousands of bytes long.

The more checked paths, the better: To reach the exploitable `fprintf` bug in the example, MAYHEM needs to reason through the loop, read input, fork a new interpreter for every possible path and check for errors. Without careful resource management, an engine can get bogged down with too many symbolic execution threads because of the huge number of possible execution paths.

Execute as much natively as possible: Symbolic execution is slow compared to concrete execution since the semantics of an instruction are simulated in software. In `orzHttpd`, millions of instructions set up the basic server before an attacker can even connect to a socket. We want to execute these instructions concretely and then switch to symbolic execution.

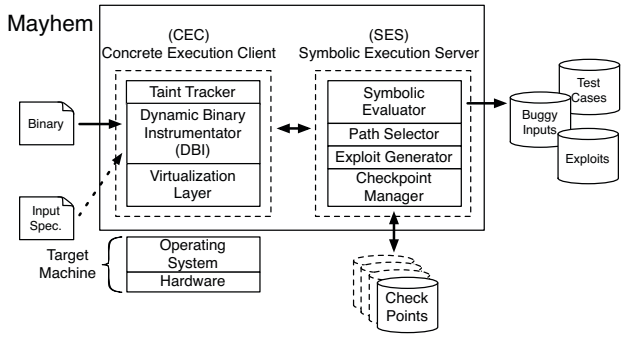


Figure 2: MAYHEM architecture

The MAYHEM architecture for finding exploitable bugs is shown in Figure 2. The user starts MAYHEM by running:

```
mayhem -sym-net 80 400 ./orzhttpd
```

The command-line tells MAYHEM to symbolically execute `orzHttpd`, and open sockets on port 80 to receive symbolic 400-byte long packets. All remaining steps to create an exploit are performed automatically.

MAYHEM consists of two concurrently running processes: a *Concrete Executor Client* (CEC), which executes code natively on a CPU, and a *Symbolic Executor Server* (SES). Both are shown in Figure 2. At a high level, the CEC runs on a target system, and the SES runs on any platform, waiting for connections from the CEC. The CEC takes in a binary program along with the potential symbolic sources (input specification) as an input, and begins communication with the SES. The SES then symbolically executes blocks that the CEC sends, and outputs several types of test cases including normal test cases, crashes, and exploits. The steps followed by MAYHEM to find the vulnerable code and generate an exploit are:

1) The `--sym-net 80 400` argument tells MAYHEM to perform symbolic execution on data read in from a socket on port 80. Effectively this is specifying which input sources are potentially under attacker control. MAYHEM can handle attacker input from environment variables, files, and the network.
2) The CEC loads the vulnerable program and connects to the SES to initialize all symbolic input sources. After the initialization, MAYHEM executes the binary concretely on the CPU in the CEC. During execution, the CEC instruments the code and performs dynamic taint analysis [23]. Our taint tracking engine checks if a block contains tainted instructions, where a block is a sequence of instructions that ends with a conditional jump or a call instruction.
3) When the CEC encounters a tainted branch condition or jump target, it suspends concrete execution. A tainted jump means that the target may be dependent on attacker

input. The CEC sends the instructions to the SES and the SES determines which branches are feasible. The CEC will later receive the next branch target to explore from the SES.

4) The SES, running in parallel with the CEC, receives a stream of tainted instructions from the CEC. The SES jits the instructions to an intermediate language (§III), and symbolically executes the corresponding IL. The CEC provides any concrete values whenever needed, e.g., when an instruction operates on a symbolic operand and a concrete operand. The SES maintains two types of formulas:
   Path Formula: The path formula reflects the constraints to reach a particular line of code. Each conditional jump adds a new constraint on the input. For example, lines 32-33 create two new paths: one which is constrained so that the read input ends in an `\r\n\r\n` and line 35 is executed, and one where the input does not end in `\r\n\r\n` and line 28 will be executed.
   Exploitability Formula: The exploitability formula determines whether i) the attacker can gain control of the instruction pointer, and ii) execute a payload.
5) When MAYHEM hits a tainted branch point, the SES decides whether we need to fork execution by querying the SMT solver. If we need to fork execution, all the new forks are sent to the path selector to be prioritized. Upon picking a path, the SES notifies the CEC about the change and the corresponding execution state is restored. If the system resource cap is reached, then the checkpoint manager starts generating checkpoints instead of forking new executors (§IV). At the end of the process, test cases are generated for the terminated executors and the SES informs the CEC about which checkpoint should continue execution next.
6) During the execution, the SES switches context between executors and the CEC checkpoints/restores the provided execution state and continues execution. To do so, the CEC maintains a virtualization layer to handle the program interaction with the underlying system and checkpoint/restore between multiple program execution states (§IV-C).
7) When MAYHEM detects a tainted jump instruction, it builds an exploitability formula, and queries an SMT solver to see if it is satisfiable. A satisfying input will be, by construction, an exploit. If no exploit is found on the tainted branch instruction, the SES keeps exploring execution paths.
8) The above steps are performed at each branch until an exploitable bug is found, MAYHEM hits a user-specified maximum runtime, or all paths are exhausted.

III. BACKGROUND

Binary Representation in our language. Basic symbolic execution is performed on assembly instructions as they execute. In the overall system the stream comes from the CEC as explained earlier; here we assume they are simply given to us. We leverage BAP [15], an open-source binary analysis framework to convert x86 assembly to an intermediate language suitable for symbolic execution. For each instruction executed, the symbolic executor jits the instruction to the BAP IL. The SES performs symbolic execution directly on the IL, introduces additional constraints related to specific attack payloads, and sends the formula to an SMT solver to check satisfiability. For example, the IL for a `ret` instruction consists of two statements: one that loads an address from memory, and one that jumps to that address.

Symbolic Execution on the IL. In concrete execution, the program is given a concrete value as input, it executes statements to produce new values, and terminates with final values. In symbolic execution we do not restrict execution to a single value, but instead provide a symbolic input variable that represents the set of all possible input values. The symbolic execution engine evaluates expressions for each statement in terms of the original symbolic inputs. When symbolic execution hits a branch, it considers two possible worlds: one where the true branch target is followed and one where the false branch target is followed. It does so by forking off an interpreter for each branch and asserting in the generated formula that the branch guard must be satisfied. The final formula encapsulates all branch conditions that must be met to execute the given path, thus is called the *path formula* or *path predicate*.

In MAYHEM, each IL statement type has a corresponding symbolic execution rule. Assertions in the IL are immediately appended to the formula. Conditional jump statements create two formulas: one where the branch guard is asserted true and the true branch is followed, and one which asserts the negation of the guard and the false branch is followed. For example, if we already have formula f and execute `cjmp` e_1, e_2, e_3 where e_1 is the branch guard and e_2 and e_3 are jump targets, then we create the two formulas:

$$f \wedge e_1 \wedge FSE(path_{e_2})$$
$$f \wedge \neg e_1 \wedge FSE(path_{e_3})$$

where FSE stands for forward symbolic execution of the jump target. Due to space, we give the exact semantics in a companion paper [15], [24].

IV. HYBRID SYMBOLIC EXECUTION

MAYHEM is a hybrid symbolic execution system. Instead of running in pure online or offline execution mode, MAYHEM can alternate between modes. In this section we present the motivation and mechanics of hybrid execution.

A. Previous Symbolic Execution Systems

Offline symbolic execution—as found in systems such as SAGE [13] —requires two inputs: the target program and an initial seed input. In the first step, offline systems concretely execute the program on the seed input and record a trace. In

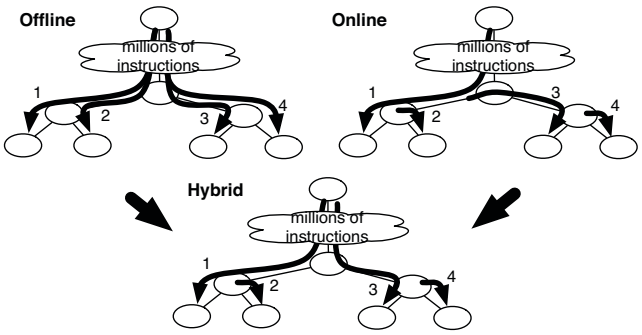


Figure 3: Hybrid execution tries to combine the speed of online execution and the memory use of offline execution to efficiently explore the input space.

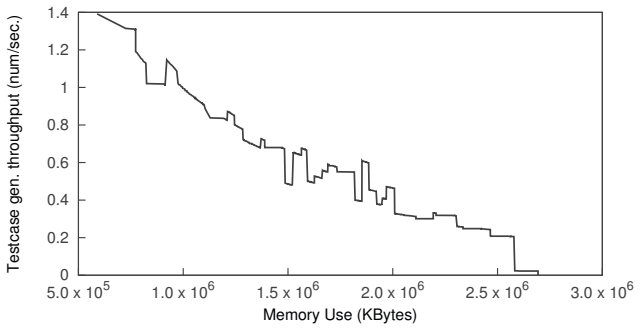


Figure 4: Online execution throughput versus memory use.

the second step, they symbolically execute the instructions in the recorded trace. This approach is called *concolic* execution, a juxtaposition of concrete and symbolic execution. Offline execution is attractive because of its simplicity and low resource requirements; we only need to handle a single execution path at a time.

The top-left diagram of Figure 3 highlights an immediate drawback of this approach. For every explored execution path, we need to first re-execute a (potentially) very large number of instructions until we reach the symbolic condition where execution forked, and then begin to explore new instructions.

Online symbolic execution avoids this re-execution cost by forking two interpreters at branch points, each one having a copy of the current execution state. Thus, to explore a different path, online execution simply needs to perform a *context switch* to the execution state of a suspended interpreter. S2E [28], KLEE [9] and AEG [2] follow this approach by performing online symbolic execution on LLVM bytecode.

However, forking off a new executor at each branch can quickly strain the memory, causing the entire system to grind to a halt. State-of-the-art online executors try to address this problem with aggressive copy-on-write optimizations. For example, KLEE has an immutable state representation and S2E shares common state between snapshots of physical memory and disks. Nonetheless, since all execution states are kept in memory simultaneously, eventually all online executors will reach the memory cap. The problem can be mitigated by using DFS (Depth-First-Search)—however, this is not a very useful strategy in practice. To demonstrate the problem, we downloaded S2E [28] and ran it on a coreutils application (`echo`) with 2 symbolic arguments, each one 10 bytes long. Figure 4 shows how the symbolic execution throughput (number of test cases generated per second) is slowed down as the memory use increases.

B. Hybrid Symbolic Execution

MAYHEM introduces *hybrid symbolic execution* to actively manage memory without constantly re-executing the same instructions. Hybrid symbolic execution alternates between online and offline modes to maximize the effectiveness of each mode. MAYHEM starts analysis in online mode. When the system reaches a memory cap, it switches to offline mode and does not fork any more executors. Instead, it produces checkpoints to start new online executions later on. The crux of the system is to distribute the online execution tasks into subtasks without losing potentially interesting paths. The hybrid execution algorithm employed by MAYHEM is split into four main phases:

1. Initialization: The first time MAYHEM is invoked for a program, it initializes the checkpoint manager, the checkpoint database, and test case directories. It then starts online execution of the program and moves to the next phase.

2. Online Exploration: During the online phase, MAYHEM symbolically executes the program in an online fashion, context-switching between current active execution states, and generating test cases.

3. Checkpointing: The checkpoint manager monitors online execution. Whenever the memory utilization reaches a cap, or the number of running executors exceeds a threshold, it will select and generate a checkpoint for an active executor. A checkpoint contains the symbolic execution state of the suspended executor (path predicate, statistics, etc.) and replay information[1]. The concrete execution state is discarded. When the online execution eventually finishes all active execution paths, MAYHEM moves to the next phase.

4. Checkpoint Restoration: The checkpoint manager selects a checkpoint based on a ranking heuristic IV-D and restores it in memory. Since the symbolic execution state was saved in the checkpoint, MAYHEM only needs to re-construct the concrete execution state. To do so, MAYHEM concretely executes the program using one satisfiable assignment of the path predicate as input, until the program reaches the instruction when the execution state was suspended. At that point, the concrete state is restored and the online exploration (phase 2) restarts. Note that phase 4 avoids symbolically re-executing instructions during the checkpoint restoration phase

[1]Note that the term "checkpoint" differs from an offline execution "seed", which is just a concrete input.

(unlike standard concolic execution), and the re-execution happens concretely. Figure 3 shows the intuition behind hybrid execution. We provide a detailed comparison between online, offline, and hybrid execution in §VIII-C.

C. Design and Implementation of the CEC

The CEC takes in the binary program, a list of input sources to be considered symbolic, and an optional checkpoint input that contains execution state information from a previous run. The CEC concretely executes the program, hooks input sources and performs taint analysis on input variables. Every basic block that contains tainted instructions is sent to the SES for symbolic execution. As a response, the CEC receives the address of the next basic block to be executed and whether to save the current state as a restoration point. Whenever an execution path is complete, the CEC context-switches to an unexplored path selected by the SES and continues execution. The CEC terminates only if all possible execution paths have been explored or a threshold is reached. If we provide a checkpoint, the CEC first executes the program concretely until the checkpoint and then continues execution as before.

Virtualization Layer. During an online execution run, the CEC handles multiple concrete execution states of the analyzed program simultaneously. Each concrete execution state includes the current register context, memory and OS state (the OS state contains a snapshot of the virtual filesystem, network and kernel state). Under the guidance of the SES and the path selector, the CEC context switches between different concrete execution states depending on the symbolic executor that is currently active. The virtualization layer mediates all system calls to the host OS and emulates them. Keeping separate copies of the OS state ensures there are no side-effects across different executions. For instance, if one executor writes a value to a file, this modification will only be visible to the current execution state—all other executors will have a separate instance of the same file.

Efficient State Snapshot. Taking a full snapshot of the concrete execution state at every fork is very expensive. To mitigate the problem, CEC shares state across execution states–similar to other systems [9], [28]. Whenever execution forks, the new execution state reuses the state of the parent execution. Subsequent modifications to the state are recorded in the current execution.

D. Design and Implementation of the SES

The SES manages the symbolic execution environment and decides which paths are executed by the CEC. The environment consists of a symbolic executor for each path, a path selector which determines which feasible path to run next, and a checkpoint manager.

The SES caps the number of symbolic executors to keep in memory. When the cap is reached, MAYHEM stops generating new interpreters and produces *checkpoints*; execution states that will explore program paths that MAYHEM was unable to explore in the first run due to the memory cap. Each checkpoint is prioritized and used by MAYHEM to continue exploration of these paths at a subsequent run. Thus, when all pending execution paths terminate, MAYHEM selects a new checkpoint and continues execution—until all checkpoints are consumed and MAYHEM exits.

Each symbolic executor maintains two contexts (as state): a variable context containing all symbolic register values and temporaries, and a memory context keeping track of all symbolic data in memory. Whenever execution forks, the SES clones the current symbolic state (to keep memory low, we keep the execution state immutable to take advantage of copy-on-write optimizations—similar to previous work [9], [28]) and adds a new symbolic executor to a priority queue. This priority queue is regularly updated by our path selector to include the latest changes (e.g., which paths were explored, instructions covered, and so on).

Preconditioned Symbolic Execution: MAYHEM implements preconditioned symbolic execution as in AEG [2]. In preconditioned symbolic execution, a user can optionally give a partial specification of the input, such as a prefix or length of the input, to reduce the range of search space. If a user does not provide a precondition, then SES tries to explore all feasible paths. This corresponds to the user providing the minimum amount of information to the system.

Path Selection: MAYHEM applies path prioritization heuristics—as found in systems such as SAGE [13] and KLEE [9]—to decide which path should be explored next. Currently, MAYHEM uses three heuristic ranking rules: a) executors exploring new code (e.g., instead of executing known code more times) have high priority, b) executors that identify symbolic memory accesses have higher priority, and c) execution paths where symbolic instruction pointers are detected have the highest priority. The heuristics are designed to prioritize paths that are most likely to contain a bug. For instance, the first heuristic relies on the assumption that previously explored code is less likely to contain a bug than new code.

E. Performance Tuning

MAYHEM employs several optimizations to speed-up symbolic execution. We present three optimizations that were most effective: 1) independent formula, 2) algebraic simplifications, and 3) taint analysis.

Similar to KLEE [9], MAYHEM splits the path predicate to independent formulas to optimize solver queries. A small implementation difference compared to KLEE is that MAYHEM keeps a map from input variables to formulas at all times. It is not constructed only for querying the solver (this representation allows more optimizations §V). MAYHEM also applies other standard optimizations as proposed by previous systems such as the constraint subsumption optimization [13], a counter-example cache [9] and others. MAYHEM also

simplifies symbolic expressions and formulas by applying algebraic simplifications, e.g. `x` $\oplus$ `x = 0`, `x & 0 = 0`, and so on.

Recall from §IV-C, MAYHEM uses taint analysis [11], [23] to selectively execute instruction blocks that deal with symbolic data. This optimization gives a $8\times$ speedup on average over executing all instruction blocks (see §VIII-G).

V. INDEX-BASED MEMORY MODELING

MAYHEM introduces an *index-based memory model* as a practical approach to handling symbolic memory loads. The index-based model allows MAYHEM to adapt its treatment of symbolic memory based on the value of the index. In this section we present the entire memory model of MAYHEM.

MAYHEM models memory as a map $\mu : I \rightarrow E$ from 32-bit indices (i) to expressions (e). In a `load(`μ`,`i`)` expression, we say that index i *indexes* memory μ, and the loaded value e represents the *contents* of the i^{th} memory cell. A load with a concrete index i is directly translated by MAYHEM into an appropriate lookup in μ (i.e., $\mu[i]$). A `store(`μ`, `i`, `e`)` instruction results in a new memory $\mu[i \leftarrow e]$ where i is mapped to e.

A. Previous Work & Symbolic Index Modeling

A symbolic index occurs when the index used in a memory lookup is not a number, but an expression—a pattern that appears very frequently in binary code. For example, a C `switch(c)` statement is compiled down to a jump-table lookup where the input character `c` is used as the index. Standard string conversion functions (such as ASCII to Unicode and vice versa, `to_lower`, `to_upper`, etc.) are all in this category.

Handling arbitrary symbolic indices is notoriously hard, since a symbolic index may (in the worst case) reference *any* cell in memory. Previous research and state-of-the-art tools indicate that there are two main approaches for handling a symbolic index: a) concretizing the index and b) allowing memory to be fully symbolic.

First, concretizing means instead of reasoning about all possible values that could be indexed in memory, we *concretize* the index to a single specific address. This concretization can reduce the complexity of the produced formulas and improve solving/exploration times. However, constraining the index to a single value may cause us to miss paths—for instance, if they depend on the value of the index. Concretization is the natural choice for offline executors, such as SAGE [13] or BitBlaze [5], since only a single memory address is accessed during concrete execution.

Reasoning about all possible indices is also possible by treating memory as fully symbolic. For example, tools such as McVeto [27], BAP [15] and BitBlaze [5] offer capabilities to handle symbolic memory. The main tradeoff—when compared with the concretization approach—is performance. Formulas involving symbolic memory are more expressive, thus solving/exploration times are usually higher.

B. Memory Modeling in MAYHEM

The first implementation of MAYHEM followed the simple concretization approach and concretized all memory indices. This decision proved to be severely limiting in that selecting a single address for the index usually did not allow us to satisfy the exploit payload constraints. Our experiments show that 40% of the examples require us to handle symbolic memory—simple concretization was insufficient (see §VIII).

The alternative approach was symbolic memory. To avoid the scalability problems associated with fully symbolic memory, MAYHEM models memory *partially*, where writes are always concretized, but symbolic reads are allowed to be modeled symbolically. In the rest of this section we describe the index-based memory model of MAYHEM in detail, as well as some of the key optimizations.

Memory Objects. To model symbolic reads, MAYHEM introduces *memory objects*. Similar to the global memory μ, a memory object $\mathcal{M}$ is also a map from 32-bit indices to expressions. Unlike the global memory however, a memory object is immutable. Whenever a symbolic index is used to read memory, MAYHEM generates a fresh memory object $\mathcal{M}$ that contains all values that could be accessed by the index—$\mathcal{M}$ is a partial snapshot of the global memory.

Using the memory object, MAYHEM can reduce the evaluation of a `load(`μ`, `i`)` expression to $\mathcal{M}[i]$. Note, that this is semantically equivalent to returning $\mu[i]$. The key difference is in the size of the symbolic array we introduce in the formula. In most cases, the memory object $\mathcal{M}$ will be orders of magnitude smaller than the entire memory μ.

Memory Object Bounds Resolution. Instantiating the memory object requires MAYHEM to find all possible values of a symbolic index i. In the worst case, this may require up to 2^{32} queries to the solver (for 32-bit memory addresses). To tackle this problem MAYHEM exchanges some accuracy for scalability by resolving the bounds $[\mathcal{L},\mathcal{U}]$ of the memory region—where $\mathcal{L}$ is the lower and $\mathcal{U}$ is the upper bound of the index. The bounds need to be conservative, i.e., all possible values of the index should be within the $[\mathcal{L},\mathcal{U}]$ interval. Note that the memory region does not need to be continuous, for example i might have only two realizable values ($\mathcal{L}$ and $\mathcal{U}$).

To obtain these bounds MAYHEM uses the solver to perform binary search on the value of the index in the context of the current path predicate. For example, initially for the lowest bound of a 32-bit i: $\mathcal{L} \in [0, 2^{32} - 1]$. If $i < \frac{2^{32}-1}{2}$ is satisfiable then $\mathcal{L} \in [0, \frac{2^{32}-1}{2} - 1]$ while unsatisfiability indicates that $\mathcal{L} \in [\frac{2^{32}-1}{2}, 2^{32} - 1]$. We repeat the process until we recover both bounds. Using the bounds we can now instantiate the memory object (using a fresh symbolic array $\mathcal{M}$) as follows: $\forall i \in [\mathcal{L},\mathcal{U}] : \mathcal{M}[i] = \mu[i]$.

The bounds resolution algorithm described above is sufficient to generate a conservative representation of memory objects and allow MAYHEM to reason about symbolic memory reads. In the rest of the section we detail the main

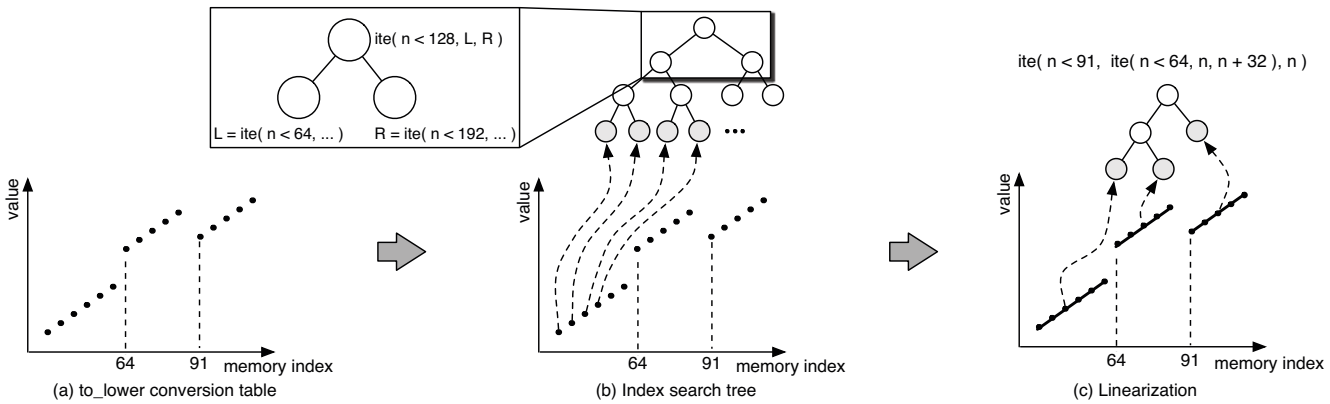


Figure 5: Figure (a) shows the `to_lower` conversion table, (b) shows the generated IST, and (c) the IST after linearization.

optimization techniques MAYHEM includes to tackle some of the caveats of the original algorithm:

- Querying the solver on every symbolic memory dereference is expensive. Even with binary search, identifying both bounds of a 32-bit index required ~ 54 queries on average (§VIII) (§V-B1,§V-B2,§V-B3).
- The memory region may not be continuous. Even though many values between the bounds may be infeasible, they are still included in the memory object, and consequently, in the formula (§V-B2).
- The values within the memory object might have structure. By modeling the object as a single byte array we are missing opportunities to optimize our formulas based on the structure. (§V-B4,§V-B5).
- In the worst case, a symbolic index may access any possible location in memory (§V-C).

1) Value Set Analysis (VSA): MAYHEM employs an online version of VSA [4] to reduce the solver load when resolving the bounds of a symbolic index (i). VSA returns a strided interval for the given symbolic index. A strided interval represents a set of values in the form $\mathcal{S}[\mathcal{L},\mathcal{U}]$, where $\mathcal{S}$ is the stride and $\mathcal{L}, \mathcal{U}$ are the bounds. For example, the interval $2[1,5]$ represents the set $\{1,3,5\}$. The strided interval output by VSA will be an over-approximation of all possible values the index might have. For instance, $i = (1 + byte) << 1$ — where $byte$ is a symbolic byte with an interval $1[0,255]$ — results in an interval: $VSA(i) = 2[2,512]$.

The strided interval produced by VSA is then refined by the solver (using the same binary-search strategy) to get the tight lower and upper bounds of the memory object. For instance, if the path predicate asserts that $byte < 32$, then the interval for the index $(1 + byte) << 1$ can be refined to $2[2,64]$. Using VSA as a preprocessing step has a cascading effect on our memory modeling: a) we perform 70% less queries to resolve the exact bounds of the memory object (§VIII), b) the strided interval can be used to eliminate impossible values in the $[\mathcal{L},\mathcal{U}]$ region, thus making formulas simpler, and c) the elimination can trigger other optimizations (see §V-B5).

2) Refinement Cache: Every VSA interval is refined using solver queries. The refinement process can still be expensive (for instance, the over-approximation returned by VSA might be too coarse). To avoid repeating the process for the same intervals, MAYHEM keeps a cache mapping intervals to potential refinements. Whenever we get a cache hit, we query the solver to check whether the cached refinement is accurate for the current symbolic index, before resorting to binary-search for refinement. The refinement cache can reduce the number of bounds-resolution queries by 82% (§VIII).

3) Lemma Cache: Checking an entry of the refinement cache still requires solver queries. MAYHEM uses another level of caching to avoid repeatedly querying α-equivalent formulas, i.e., formulas that are structurally equivalent up to variable renaming. To do so, MAYHEM converts queried formulas to a canonical representation (F) and caches the query results (Q) in the form of a *lemma*: $F \rightarrow Q$. The answer for any formula mapping to the same canonical representation is retrieved immediately from the cache. The lemma cache can reduce the number of bounds-resolution queries by up to 96% (§VIII). The effectiveness of this cache depends on the independent formulas optimization §IV-E. The path predicate has to be represented as a set of independent formulas, otherwise any new formula addition to the current path predicate would invalidate all previous entries of the lemma cache.

4) Index Search Trees (ISTs): Any value loaded from a memory object $\mathcal{M}$ is symbolic. To resolve constraints involving a loaded value ($\mathcal{M}[i]$), the solver needs to both find an entry in the object that satisfies the constraints *and* ensure that the index to the object entry is realizable. To lighten the burden on the solver, MAYHEM replaces memory object lookup expressions with *index search trees (ISTs)*. An IST is a binary search tree where the symbolic index is the key and the leaf nodes contain the entries of the object. The entire tree is encoded in the formula representation of the load expression.

More concretely, given a (sorted by address) list of

entries E within a memory object $\mathcal{M}$, a balanced IST for a symbolic index i is defined as: $IST(E) = ite(i < addr(E_{right}), E_{left}, E_{right}))$, where `ite` represents an if-then-else expression, E_{left} (E_{right}) represents the left (right) half of the initial entries E, and $addr(\cdot)$ returns the lowest address of the given entries. For a single entry the IST returns the entry without constructing any `ite` expressions.

Note that the above definition constructs a balanced IST. We could instead construct the IST with nested `ite` expressions—making the formula depth $O(n)$ in the number of object entries instead of $O(\log n)$. However, our experimental results show that a balanced IST is $4\times$ faster than a nested IST (§VIII). Figure 5 shows how MAYHEM constructs the IST when given the entries of a memory object (the `to_lower` conversion table) with a single symbolic character as the index.

5) Bucketization with Linear Functions: The IST generation algorithm creates a leaf node for each entry in the memory object. To reduce the number of entries, MAYHEM performs an extra preprocessing step before passing the object to the IST. The idea is that we can use the memory object structure to combine multiple entries into a single *bucket*. A bucket is an index-parameterized expression that returns the value of the memory object for every index within a range.

MAYHEM uses linear functions to generate buckets. Specifically, MAYHEM sweeps all entries within a memory object and joins consecutive points ($\langle index, value\rangle$ tuples) into lines, a process we call *linearization*. Any two points can form a line $y = \alpha x + \beta$. Follow-up points $\langle i_i, v_i\rangle$ will be included in the same line if $u_i = \alpha i_i + \beta$. At the end of linearization, the memory object is split into a list of buckets, where each bucket is either a line or an isolated point. The list of buckets can now be passed to the IST algorithm. Figure 5 shows the `to_lower` IST after applying linearization. Linearization effectively reduces the number of leaf nodes from 256 to 3.

The idea of using linear functions to simplify memory lookups comes from a simple observation: linear-like patterns appear frequently for several operations at the binary level. For example, jump tables generated by switch statements, conversion and translation tables (e.g., ASCII to Unicode and vice versa) all contain values that are scaling linearly with the index.

C. Prioritized Concretization.

Modeling a symbolic load using a memory object is beneficial when the size of the memory object is significantly smaller than the entire memory ($|\mathcal{M}| \ll |\mu|$). Thus, the above optimizations are only activated when the size of the memory object, approximated by the range, is below a threshold ($|\mathcal{M}| < 1024$ in our experiments).

Whenever the memory object size exceeds the threshold, MAYHEM will concretize the index used to access it. However, instead of picking a satisfying value at random, MAYHEM attempts to *prioritize* the possible concretization values. Specifically, for every symbolic pointer, MAYHEM performs three checks:

1) Check if it is possible to redirect the pointer to unmapped memory under the context of the current path predicate. If true, MAYHEM will generate a crash test case for the satisfying value.
2) Check if it is possible to redirect the symbolic pointer to symbolic data. If it is, MAYHEM will redirect (and concretize) the pointer to the least constrained region of the symbolic data. By redirecting the pointer towards the least constrained region, MAYHEM tries to avoid loading overconstrained values, thus eliminating potentially interesting paths that depend on these values. To identify the least constrained region, MAYHEM splits memory into symbolic regions, and sorts them based on the complexity of constraints associated with each region.
3) If all of the above checks fail, MAYHEM concretizes the index to a valid memory address and continues execution.

The above steps infer whether a symbolic expression is a pointer, and if so, whether it is valid or not (e.g., NULL). For example, Figure 6 contains a buffer overflow at line 9. However, an attacker is not guaranteed to hijack control even if `strcpy` overwrites the return address. The program needs to reach the return instruction to actually transfer control. However, at line 10 the program performs two dereferences both of which need to succeed (i.e., avoid crashing the program) to reach line 11 (note that pointer ptr is already overwritten with user data). MAYHEM augmented with prioritized concretization will generate 3 distinct test cases: 1) a crash test case for an invalid dereference of pointer ptr, 2) a crash test case where dereferencing pointer bar fails after successfully redirecting ptr to symbolic data, and 3) an exploit test case, where both dereferences succeed and user input hijacks control of the program. Figure 6 shows the memory layout for the third test case.

```
1  typedef struct {
2    int value;
3    char * bar;
4  } foo;
5  int vulnerable (char * input)
6  {
7    foo * ptr = init;
8    buffer[100];
9    strcpy(buffer, input);
10   buffer[0] = ptr->bar[0];
11   return 0;
12 }
```

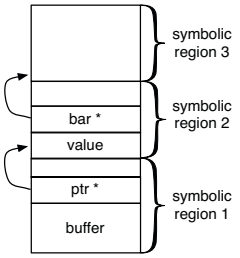


Figure 6: MAYHEM reconstructing symbolic data structures.

VI. EXPLOIT GENERATION

MAYHEM checks for two exploitable properties: a symbolic (tainted) instruction pointer, and a symbolic format string. Each property corresponds to a buffer overflow and format string attack respectively. Whenever any of the two

exploitable policies are violated, MAYHEM generates an exploitability formula and tries to find a satisfying answer, i.e., an exploit.

MAYHEM can generate both local and remote attacks. Our generic design allows us to handle both types of attacks similarly. For Windows, MAYHEM detects overwritten Structured Exception Handler (SEH) on the stack when an exception occurs, and tries to create an SEH-based exploit.

Buffer Overflows: MAYHEM generates exploits for any possible instruction-pointer overwrite, commonly triggered by a buffer overflow. When MAYHEM finds a symbolic instruction pointer, it first tries to generate jump-to-register exploits, similar to previous work [14]. For this type of exploit, the instruction pointer should point to a trampoline, e.g. `jmp %eax`, and the register, e.g. `%eax`, should point to a place in memory where we can place our shellcode. By encoding those constraints into the formula, MAYHEM is able to query the solver for a satisfying answer. If an answer exists, we proved that the bug is exploitable. If we can't generate a jump-to-register exploit, we try to generate a simpler exploit by making the instruction pointer point directly to a place in memory where we can place shellcode.

Format String Attacks: To identify and generate format string attacks, MAYHEM checks whether the format argument of format string functions, e.g., `printf`, contains any symbolic bytes. If any symbolic bytes are detected, it tries to place a format string payload within the argument that will overwrite the return address of the formatting function.

VII. IMPLEMENTATION

MAYHEM consists of about 27,000 lines of C/C++ and OCaml code. Our binary instrumentation framework was built on Pin [18] and all the hooks for modeled system and API calls were written in C/C++. The symbolic execution engine is written solely in OCaml and consists of about 10,000 lines of code. We rely on BAP [15] to convert assembly instructions to the IL. We use Z3 [12] as our decision procedure, for which we built direct OCaml bindings. To allow for remote communication between the two components we implemented our own cross-platform, light-weight RPC protocol (both in C++ and OCaml). Additionally, to compare between different symbolic execution modes, we implemented all three: online, offline and hybrid.

VIII. EVALUATION

A. Experimental Setup

We evaluated our system on 2 virtual machines running on a desktop with a 3.40GHz Intel(R) Core i7-2600 CPU and 16GB of RAM. Each VM had 4GB RAM and was running Debian Linux (Squeeze) VM and Windows XP SP3 respectively.

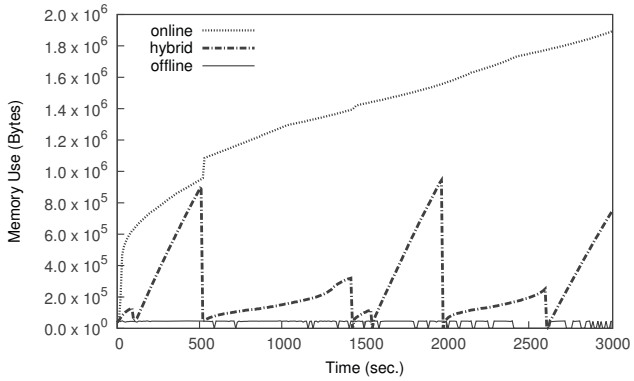


Figure 7: Memory use in online, offline, and hybrid mode.

B. Exploitable Bug Detection

We downloaded 29 different vulnerable programs to check the effectiveness of MAYHEM. Table I summarizes our results. Experiments were performed on stripped unmodified binaries on both Linux and Windows. One of the Windows applications MAYHEM exploited (`Dizzy`) was a packed binary.

Column 3 shows the type of exploits that MAYHEM detected as we described in §VI. Column 4 shows the symbolic sources that we considered for each program. There are examples from all the symbolic input sources that MAYHEM supports, including command-line arguments (Arg.), environment variables (Env. Vars), network packets (Network) and symbolic files (Files). Column 5 is the size of each symbolic input. Column 6 describes the precondition types that we provided to MAYHEM, for each of the 29 programs. They are split into three categories: length, prefix and crashing input as described in §IV-D. Column 7 shows the advisory reports for all the demonstrated exploits. In fact, MAYHEM found 2 zero-day exploits for two Linux applications, both of which we reported to the developers.

The last column contains the exploit generation time for the programs that MAYHEM analyzed. We measured the exploit generation time as the time taken from the start of analysis until the creation of the first working exploit. The time required varies greatly with the complexity of the application and the size of symbolic inputs. The fastest program to exploit was the Linux wireless configuration utility `iwconfig` in 1.90 seconds and the longest was the Windows program `Dizzy`, which took about 4 hours.

C. Scalability of Hybrid Symbolic Execution

We measured the effectiveness of hybrid symbolic execution across two scaling dimensions: memory use and speed.

Less Memory-Hungry than Online Execution. Figure 7 shows the average memory use of MAYHEM over time while analyzing a utility in coreutils (`echo`) with online, offline and hybrid execution. After a few minutes, online

	Program	Exploit Type	Input Source	Symbolic Input Size	Symb. Mem.	Precondition	Advisory ID.	Exploit Gen. Time (s)
Linux	**A2ps**	Stack Overflow	Env. Vars	550		crashing	EDB-ID-816	189
	Aeon	Stack Overflow	Env. Vars	1000		length	CVE-2005-1019	10
	Aspell	Stack Overflow	Stdin	750		crashing	CVE-2004-0548	82
	Atphttpd	Stack Overflow	Network	800	✓	crashing	CVE-2000-1816	209
	FreeRadius	Stack Overflow	Env.	9000		length	Zero-Day	133
	GhostScript	Stack Overflow	Arg.	2000		prefix	CVE-2010-2055	18
	Glftpd	Stack Overflow	Arg.	300		length	OSVDB-ID-16373	4
	Gnugol	Stack Overflow	Env.	3200		length	Zero-Day	22
	Htget	Stack Overflow	Env. vars	350	✓	length	N/A	7
	Htpasswd	Stack Overflow	Arg.	400	✓	prefix	OSVDB-ID-10068	4
	Iwconfig	Stack Overflow	Arg.	400		length	CVE-2003-0947	2
	Mbse-bbs	Stack Overflow	Env. vars	4200	✓	length	CVE-2007-0368	362
	nCompress	Stack Overflow	Arg.	1400		length	CVE-2001-1413	11
	OrzHttpd	Format String	Network	400		length	OSVDB-ID-60944	6
	PSUtils	Stack Overflow	Arg.	300		length	EDB-ID-890	46
	Rsync	Stack Overflow	Env. Vars	100	✓	length	CVE-2004-2093	8
	SharUtils	Format String	Arg.	300		prefix	OSVDB-ID-10255	17
	Socat	Format String	Arg.	600		prefix	CVE-2004-1484	47
	Squirrel Mail	Stack Overflow	Arg.	150		length	CVE-2004-0524	2
	Tipxd	Format String	Arg.	250		length	OSVDB-ID-12346	10
	xGalaga	Stack Overflow	Env. Vars	300		length	CVE-2003-0454	3
	Xtokkaetama	Stack Overflow	Arg.	100		crashing	OSVDB-ID-2343	10
Windows	**Coolplayer**	Stack Overflow	Files	210	✓	crashing	CVE-2008-3408	164
	Destiny	Stack Overflow	Files	2100	✓	crashing	OSVDB-ID-53249	963
	Dizzy	Stack Overflow (SEH)	Arg.	519	✓	crashing	EDB-ID-15566	13,260
	GAlan	Stack Overflow	Files	1500	✓	prefix	OSVDB-ID-60897	831
	GSPlayer	Stack Overflow	Files	400	✓	crashing	OSVDB-ID-69006	120
	Muse	Stack Overflow	Files	250	✓	crashing	OSVDB-ID-67277	481
	Soritong	Stack Overflow (SEH)	Files	1000	✓	crashing	CVE-2009-1643	845

Table I: List of programs that MAYHEM demonstrated as exploitable.

execution reaches the maximum number of live interpreters and starts terminating execution paths. At this point, the memory keeps increasing linearly as the paths we explore become deeper. Note that at the beginning, hybrid execution consumes as much memory as online execution without exceeding the memory threshold, and utilizes memory resources more aggressively than offline execution throughout the execution. Offline execution requires much less memory (less than 500KB on average), but at a performance cost, as demonstrated below.

Faster than Offline Execution. Figure 8 shows the exploration time for `/bin/echo` using different limits on the maximum number of running executors. For this experiment, we use 6 bytes of symbolic arguments to explore the entire input space in a reasonable amount of time. When the maximum number of running executors is 1, it means

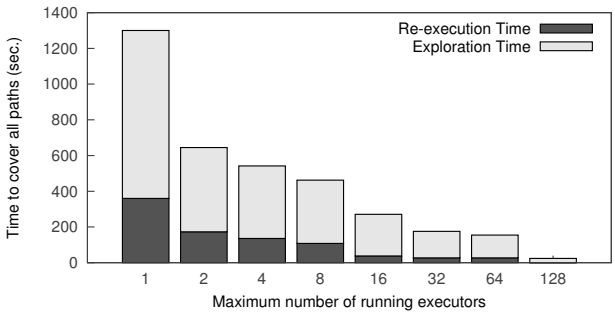


Figure 8: Exploration times for different limits on the maximum number of running executors.

MAYHEM will produce a disk checkpoint—the average checkpoint size was 30KB—for every symbolic branch,

	L Hits	R Hits	Misses	# Queries	Time (sec)
No opt.	N/A	N/A	N/A	217,179	1,841
+ VSA	N/A	N/A	N/A	49,424	437
+ R cache	N/A	3996	7	10,331	187
+ L cache	3940	56	7	242	77

Table II: Effectiveness of bounds resolution optimizations. The L and R caches are respectively the Lemma and Refinement caches as defined in §V.

thus is equivalent to offline execution. When the maximum number of running executors was 128 or above, MAYHEM did not have to checkpoint to disk, thus is equivalent to an online executor. As a result, online execution took around 25 seconds to explore the input space while offline execution needed 1,400 seconds. Online was 56× faster than offline in this experiment. We identified two major reasons for this performance boost.

First, the re-execution cost is higher than context-switching between two execution states (§IV-B). MAYHEM spent more than 25% of the time re-executing previous paths in the offline scheme. For the online case, 2% of the time was spent context-switching. Second, online is more cache-efficient than offline execution in our implementation. Specifically, online execution makes more efficient use of the Pin code cache [18] by switching between paths in-memory during a single execution. As a result, the code cache made online execution 40× faster than offline execution.

Additionally, we ran a Windows GUI program (`MiniShare`) to compare the throughput between offline and hybrid execution. We chose this program because it does not require user interaction (e.g., mouse click), to start symbolic execution. We ran the program for 1 hour for each execution mode. Hybrid execution was 10× faster than offline execution.

D. Handling Symbolic Memory in Real-World Applications

Recall from §V, index-based memory modeling enables MAYHEM to reason about symbolic indices. Our experiments from Table I show that more than 40% of the programs required symbolic memory modeling (column 6) to exploit. In other words, MAYHEM—after several hours of analysis—was unable to generate exploits for these programs without index-based memory modeling. To understand why, we evaluated our index-based memory modeling optimizations on the `atphttpd` server.

Bounds Resolution Table II shows the time taken by MAYHEM to find a vulnerability in atphttpd using different levels of optimizations for the bounds resolution algorithm. The times include exploit detection but not exploit generation time (since it is not affected by the bounds resolution algorithm). Row 3 shows that VSA reduces the average number of queries to the SMT solver from ∼54 to ∼14 queries per symbolic memory access, and reduces the total time by 75%.

Row 4 shows shows the number of queries when the refinement cache (R cache) is enabled on top of VSA. The R cache reduces the number of necessary binary searches to from 4003 to 7, resulting in a 57% speedup. The last row shows the effect of the lemma cache (L cache) on top of the other optimizations. The L cache takes most of the burden off the R cache, thus resulting in an additional 59% speedup. We do not expect the L cache to always be that efficient, since it relies heavily on the independence of formulas in the path predicate. The cumulative speedup was 96%.

Formula Representation	**Time (sec.)**
Unbalanced binary tree	1,754
Balanced binary tree	425
Balanced binary tree + Linearization	192

Table III: Performance comparison for different IST representations.

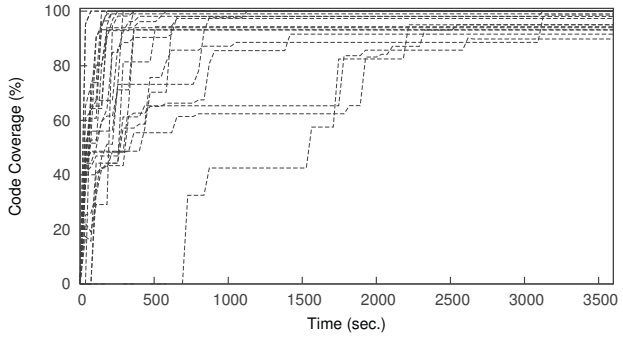


Figure 9: Code coverage achieved by MAYHEM as time progresses for 25 coreutils applications.

Index Search Tree Representation. Recall from §V-B MAYHEM models symbolic memory loads as ISTs. To show the effectiveness of this optimization we ran `atphttpd` with three different formula representations (shown in Table III). The balanced IST was more than 4× faster than the unbalanced binary tree representation, and with linearization of the formula we obtained a cumulative 9× speedup. Note, that with symbolic arrays (no ISTs) we were unable to detect an exploit within the time limit.

E. MAYHEM *Coverage Comparison*

To evaluate MAYHEM's ability to cover new paths, we downloaded an open-source symbolic executor (KLEE) to compare the performance against MAYHEM. Note KLEE runs on source, while MAYHEM on binary.

We measured the code coverage of 25 coreutils applications as a function of time. MAYHEM ran for one hour, at most, on each of those applications. We used the generated test cases to measure the code coverage using the GNU gcov

	AEG		MAYHEM			
Program	Time	LLVM	Time	ASM	Tainted ASM	Tained IL
iwconfig	0.506s	10,876	1.90s	394,876	2,200	12,893
aspell	8.698s	87,056	24.62s	696,275	26,647	133,620
aeon	2.188s	18,539	9.67s	623,684	7,087	43,804
htget	0.864s	12,776	6.76s	576,005	2,670	16,391
tipxd	2.343s	82,030	9.91s	647,498	2,043	19,198
ncompress	5.511s	60,860	11.30s	583,330	8,778	71,195

Table IV: AEG comparison: binary-only execution requires more instructions.

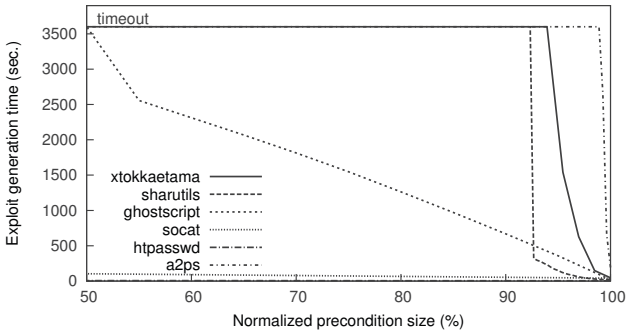


Figure 10: Exploit generation time versus precondition size.

utility. The results are shown in Figure 9.

We used the 21 tools with the smallest code size, and 4 bigger tools that we selected. MAYHEM achieved a 97.56% average coverage per application and got 100% coverage on 13 tools. For comparison, KLEE achieved 100% coverage on 12 coreutils without simulated system call failures (to have the same configuration as MAYHEM). Thus, MAYHEM seems to be competitive with KLEE for this data set. Note that MAYHEM is not designed specifically for maximizing code coverage. However, our experiments provide a rough comparison point against other symbolic executors.

F. Comparison against AEG

We picked 8 different programs from the AEG working examples [2] and ran both tools to compare exploit generation times on each of those programs using the same configuration (Table IV). MAYHEM was on average 3.4× slower than AEG. AEG uses source code, thus has the advantage of operating at a higher-level of abstraction. At the binary level, there are no types and high-level structures such as functions, variables, buffers and objects. The number of instructions executed (Table IV) is another factor that highlights the difference between source and binary-only analysis. Considering this, we believe this is a positive and competitive result for MAYHEM.

Precondition Size. As an additional experiment, we measured how the presence of a precondition affects exploit generation times. Specifically, we picked 6 programs that require a crashing input to find an exploitable bug and started to iteratively decrease the size of the precondition and measured exploit generation times. Figure 10 summarizes our results in terms of normalized precondition sizes—for example, a normalized precondition of 70% for a 100-byte crashing input means that we provide 70 bytes of the crashing input as a precondition to MAYHEM. While the behavior appeared to be program-dependent, in most of the programs we observed a sudden phase-transition, where the removal of a single character could cause MAYHEM to not detect the exploitable bug within the time limit. We believe this to be an interesting topic for future work in the area.

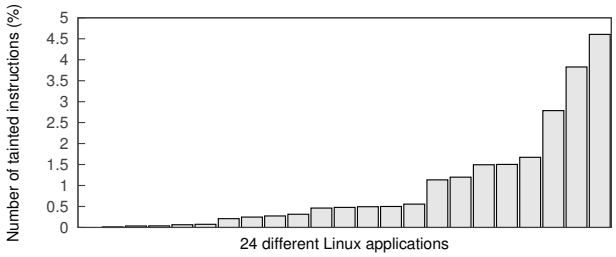


Figure 12: Tainted instructions (%) for 24 Linux applications.

G. Performance Tuning

Formula Optimizations. Recall from §IV-E MAYHEM uses various optimization techniques to make solver queries faster. To compare against our optimized version of MAYHEM, we turned off some or all of these optimizations.

We chose 15 Linux programs to evaluate the speedup obtained with different levels of optimizations turned on. Figure 11 shows the head-to-head comparison (in exploit finding and generation times) between 4 different formula optimization options. Algebraic simplifications usually speed up our analysis and offer an average speedup of 10% for the 15 test programs. Significant speedups occur when the independent formula optimization is turned on along with simplifications, offering speedups of 10-100×.

Z3 supports incremental solving, so as an additional experiment, we measured the exploit generation time with Z3 in incremental mode. In most cases solving times for incremental formulas are comparable to the times we obtain with the independent formulas optimization. In fact, in half of our examples (7 out of 15) incremental formulas outperform independent formulas. In contrast to previous results, this implies that using the solver in incremental mode can alleviate the need for many formula simplifications and optimizations. A downside of using the solver in incremental mode was that it made our symbolic execution state mutable—and thus was less memory efficient during our long-running tests.

Tainted Instructions. Only tainted instruction blocks are evaluated symbolically by MAYHEM—all other blocks are executed natively. Figure 12 shows the percentage of tainted instructions for 24 programs (taken from Table I). More than 95% of instructions were not tainted in our sample programs, and this optimization gave about 8× speedup on average.

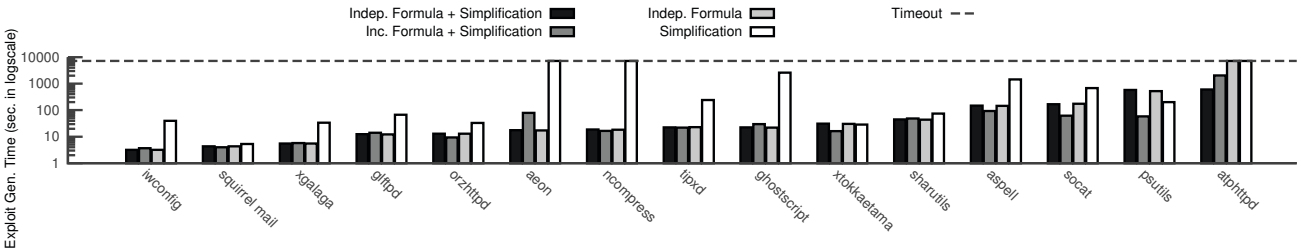


Figure 11: Exploit generation time of MAYHEM for different optimizations.

IX. DISCUSSION

Most of the work presented in this paper focuses on exploitable bug finding. However, we believe that the main techniques can be adapted to other application domains under the context of symbolic execution. We also believe that our hybrid symbolic execution and index-based memory modeling represent new points in the design space of symbolic execution.

We stress that the intention of MAYHEM is informing a user that an exploitable bug exists. The exploit produced is intended to demonstrate the severity of the problem, and to help debug and address the underlying issue. MAYHEM makes no effort to bypass OS defenses such as ASLR and DEP, which will likely protect systems against exploits we generate. However, our previous work on Q [25] shows that a broken exploit (that no longer works because of ASLR and DEP), can be automatically transformed—with high probability—into an exploit that bypasses both defenses on modern OSes. While we could feed the exploits generated by MAYHEM directly into Q, we do not explore this possibility in this paper.

Limitations: MAYHEM does not have models for all system/library calls. The current implementation models about 30 system calls in Linux, and 12 library calls in Windows. To analyze larger and more complicated programs, more system calls need to be modeled. This is an artifact of performing per-process symbolic execution. Whole-system symbolic executors such as S2E [28] or BitBlaze [5] can execute both user and kernel code, and thus do not have this limitation. The down-side is that whole-system analysis can be much more expensive, because of the higher state restoration cost and the time spent analyzing kernel code. Another limitation is that MAYHEM can currently analyze only a single execution thread on every run. MAYHEM cannot handle multi-threaded programs when threads interact with each other (through message-passing or shared memory). Last, MAYHEM executes only tainted instructions, thus it is subject to all the pitfalls of taint analysis, including undertainting, overtainting and implicit flows [24].

Future Work: Our experiments show that MAYHEM can generate exploits for standard vulnerabilities such as stack-based buffer overflows and format strings. An interesting future direction is to extend MAYHEM to handle more advanced exploitation techniques such as exploiting heap-based buffer overflows, use-after-free vulnerabilities, and information disclosure attacks. At a high level, it should be possible to detect such attacks using safety properties similar to the ones MAYHEM currently employs. However, it is still an open question how the same techniques can scale and detect such exploits in bigger programs.

X. RELATED WORK

Brumley et al. [7] introduced the automatic *patch-based* exploit generation (APEG) challenge. APEG used the patch to point out the location of the bug and then used slicing to construct a formula for code paths from input source to vulnerable line. MAYHEM finds vulnerabilities and vulnerable code paths itself. In addition, APEG's notion of an exploit is more abstract: any input that violates checks introduced by the path are considered exploits. Here we consider specifically control flow hijack exploits, which were not automatically generated by APEG.

Heelan [14] was the first to describe a technique that takes in a crashing input for a program, along with a jump register, and automatically generates an exploit. Our research explores the state space to find such crashing inputs.

AEG [2] was the first system to tackle the problem of both identifying exploitable bugs and automatically generating exploits. AEG worked solely on source code and introduced preconditioned symbolic execution as a way to focus symbolic execution towards a particular part of the search space. MAYHEM is a logical extension of AEG to binary code. In practice, working on binary code opens up automatic exploit generation to a wider class of programs and scenarios.

There are several binary-only symbolic execution frameworks such as Bouncer [10], BitFuzz [8], BitTurner [6] FuzzBall [20], McVeto [27], SAGE [13], and S2E [28], which have been used in a variety of application domains. The main question we tackle in MAYHEM is scaling to find and demonstrate exploitable bugs. The hybrid symbolic execution technique we present in this paper is completely different from hybrid concolic testing [19], which interleaves random testing with concolic execution to achieve better code coverage.

XI. Conclusion

We presented MAYHEM, a tool for automatically finding exploitable bugs in binary (i.e., executable) programs in an efficient and scalable way. To this end, MAYHEM introduces a novel hybrid symbolic execution scheme that combines the benefits of existing symbolic execution techniques (both online and offline) into a single system. We also present index-based memory modeling, a technique that allows MAYHEM to discover more exploitable bugs at the binary-level. We used MAYHEM to analyze 29 applications and automatically identified and demonstrated 29 exploitable vulnerabilities.

XII. Acknowledgements

We thank our shepherd, Cristian Cadar and the anonymous reviewers for their helpful comments and feedback. This research was supported by a DARPA grant to CyLab at Carnegie Mellon University (N11AP20005/D11AP00262), a NSF Career grant (CNS0953751), and partial CyLab ARO support from grant DAAD19-02-1-0389 and W911NF-09-1-0273. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

[1] "Orzhttpd, a small and high performance http server," http://code.google.com/p/orzhttpd/.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proc. of the Network and Distributed System Security Symposium*, Feb. 2011.

[3] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-Directed Dynamic Automated Test Generation," in *International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2011, pp. 12–22.

[4] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables." in *Proc. of the International Conference on Compiler Construction*, 2004.

[5] "BitBlaze binary analysis project," http://bitblaze.cs.berkeley.edu, 2007.

[6] BitTurner, "BitTurner," http://www.bitturner.com.

[7] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. of the IEEE Symposium on Security and Privacy*, May 2008.

[8] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proc. of the ACM Conference on Computer and Communications Security*, Chicago, IL, October 2010.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the USENIX Symposium on Operating System Design and Implementation*, Dec. 2008.

[10] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Symposium on Operating Systems Principles*, Oct. 2007.

[11] J. R. Crandall and F. Chong, "Minos: Architectural support for software security through control data integrity," in *Proc. of the International Symposium on Microarchitecture*, Dec. 2004.

[12] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.

[13] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. of the Network and Distributed System Security Symposium*, Feb. 2008.

[14] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," Oxford University, Tech. Rep. MSc Thesis, 2002.

[15] I. Jager, T. Avgerinos, E. J. Schwartz, and D. Brumley, "BAP: A binary analysis platform," in *Proc. of the Conference on Computer Aided Verification*, 2011.

[16] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 386–394, 1976.

[17] Launchpad, https://bugs.launchpad.net/ubuntu, open bugs in Ubuntu. Checked 03/04/12.

[18] 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 *Proc. of the ACM Conference on Programming Language Design and Implementation*, Jun. 2005.

[19] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. of the ACM Conference on Software Engineering*, 2007, pp. 416–426.

[20] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, Mar. 2012.

[21] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, 2007.

[22] T. Newsham, "Format string attacks," Guardent, Inc., Tech. Rep., 2000.

[23] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the Network and Distributed System Security Symposium*, Feb. 2005.

[24] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. of the IEEE Symposium on Security and Privacy*, May 2010, pp. 317–331.

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. of the USENIX Security Symposium*, 2011.

[26] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. of the ACM Symposium on the Foundations of Software Engineering*, 2005.

[27] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps, "Directed proof generation for machine code," in *CAV*, 2010, pp. 288–305.

[28] G. C. Vitaly Chipounov, Volodymyr Kuznetsov, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.