AEG: Automatic Exploit Generation

Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{thanassis, sangkilc, brentlim, dbrumley}@cmu.edu

Abstract

The automatic exploit generation challenge is given a program, automatically find vulnerabilities and generate exploits for them. In this paper we present `AEG`*, the first end-to-end system for fully automatic exploit generation. We used* `AEG` *to analyze 14 open-source projects and successfully generated 16 control flow hijacking exploits. Two of the generated exploits (expect-5.43 and htget-0.93) are zero-day exploits against unknown vulnerabilities. Our contributions are: 1) we show how exploit generation for control flow hijack attacks can be modeled as a formal verification problem, 2) we propose preconditioned symbolic execution, a novel technique for targeting symbolic execution, 3) we present a general approach for generating working exploits once a bug is found, and 4) we build the first end-to-end system that automatically finds vulnerabilities and generates exploits that produce a shell.*

1 Introduction

Control flow exploits allow an attacker to execute arbitrary code on a computer. Current state-of-the-art in control flow exploit generation is for a human to think very hard about whether a bug can be exploited. Until now, automated exploit generation where bugs are automatically found and exploits are generated has not been shown practical against real programs.

In this paper, we develop novel techniques and an end-to-end system for automatic exploit generation (AEG) on real programs. In our setting, we are given the potentially buggy program in source form. Our AEG techniques find bugs, determine whether the bug is exploitable, and, if so, produce a working control flow hijack exploit string. The exploit string can be directly fed into the vulnerable application to get a shell. We have analyzed 14 open-source projects and successfully generated 16 control flow hijacking exploits, including two zero-day exploits for previously unknown vulnerabilities.

Our automatic exploit generation techniques have several immediate security implications. First, practical AEG fundamentally changes the perceived capabilities of attackers. For example, previously it has been believed that it is relatively difficult for untrained attackers to find novel vulnerabilities and create zero-day exploits. Our research shows this assumption is unfounded. Understanding the capabilities of attackers informs what defenses are appropriate. Second, practical AEG has applications to defense. For example, automated signature generation algorithms take as input a set of exploits, and output an IDS signature (*aka* an input filter) that recognizes subsequent exploits and exploit variants [3, 8, 9]. Automated exploit generation can be fed into signature generation algorithms by defenders without requiring real-life attacks.

Challenges. There are several challenges we address to make AEG practical:

A. Source code analysis alone is inadequate and insufficient. Source code analysis is insufficient to report whether a potential bug is exploitable because errors are found with respect to source code level abstractions. Control flow exploits, however, must reason about binary and runtime-level details, such as stack frames, memory addresses, variable placement and allocation, and many other details unavailable at the source code level. For instance, consider the following code excerpt:

```
char src[12], dst[10];
strncpy(dst, src, sizeof(src));
```

In this example, we have a classic buffer overflow where a larger buffer (12 bytes) is copied into a smaller buffer (10 bytes). While such a statement is clearly wrong [1] and would be reported as a bug at the source

[1] Technically, the C99 standard would say the program exhibits undefined behavior at this point.

code level, in practice this bug would likely not be exploitable. Modern compilers would page-align the declared buffers, resulting in both data structures getting 16 bytes. Since the destination buffer would be 16 bytes, the 12-byte copy would not be problematic and the bug not exploitable.

While source code analysis is insufficient, binary-level analysis is unscalable. Source code has abstractions, such as variables, buffers, functions, and user-constructed types that make automated reasoning easier and more scalable. No such abstractions exist at the binary-level; there only stack frames, registers, gotos and a globally addressed memory region.

In our approach, we combine source-code level analysis to improve scalability in finding bugs and binary and runtime information to exploit programs. To the best of our knowledge, we are the first to combine analysis from these two very different code abstraction levels.

B. Finding the exploitable paths among an infinite number of possible paths. Our techniques for AEG employ symbolic execution, a formal verification technique that explores program paths and checks if each path is exploitable. Programs have loops, which in turn means that they have a potentially infinite number of paths. However, not all paths are equally likely to be exploitable. Which paths should we check first?

Our main focus is to detect exploitable bugs. Our results show (§ 8) that existing state-of-the-art solutions proved insufficient to detect such security-critical bugs in real-world programs.

To address the path selection challenge, we developed two novel contributions in AEG. First, we have developed *preconditioned symbolic execution*, a novel technique which targets paths that are more likely to be exploitable. For example, one choice is to explore only paths with the maximum input length, or paths related to HTTP GET requests. While preconditioned symbolic execution eliminates some paths, we still need to prioritize which paths we should explore first. To address this challenge, we have developed a priority queue *path prioritization* technique that uses heuristics to choose likely more exploitable paths first. For example, we have found that if a programmer makes a mistake—not necessarily exploitable—along a path, then it makes sense to prioritize further exploration of the path since it is more likely to eventually lead to an exploitable condition.

C. An end-to-end system. We provide the first practical end-to-end system for AEG on real programs. An end-to-end system requires not only addressing a tremendous number of scientific questions, e.g., binary program analysis and efficient formal verification, but also a tremendous number of engineering issues. Our AEG implementation is a single command line that analyzes source code programs, generates symbolic execution formulas, solves them, performs binary analysis, generates binary-level runtime constraints, and formats the output as an actual exploit string that can be fed directly into the vulnerable program. A video demonstrating the end-to-end system is available online [1].

Scope. While, in this paper, we make exploits robust against local environment changes, our goal is not to make exploits robust against common security defenses, such as address space randomization [25] and $w \oplus x$ memory pages (e.g., Windows DEP). In this work, we always require source code. AEG on binary-only is left as future work. We also do not claim AEG is a "solved" problem; there is always opportunity to improve performance, scalability, to work on a larger variety of exploit classes, and to work in new application settings.

2 Overview of AEG

This section explains how `AEG` works by stepping through the entire process of bug-finding and exploit generation on a real world example. The target application is the setuid root `iwconfig` utility from the `Wireless Tools` package (version 26), a program consisting of about 3400 lines of C source code.

Before `AEG` starts the analysis, there are two necessary preprocessing steps: 1) We build the project with the GNU C Compiler (GCC) to create the binary we want to exploit, and 2) with the LLVM [17] compiler—to produce bytecode that our bug-finding infrastructure uses for analysis. After the build, we run our tool, `AEG`, and get a control flow hijacking exploit in less than 1 second. Providing the exploit string to the `iwconfig` binary, as the 1st argument, results in a root shell. We have posted a demonstration video online [1].

Figure 1 shows the code snippet that is relevant to the generated exploit. `iwconfig` has a classic `strcpy` buffer overflow vulnerability in the `get_info` function (line 15), which `AEG` spots and exploits automatically in less than 1 second. To do so, our system goes through the following analysis steps:

1. `AEG` searches for bugs at the source code level by exploring execution paths. Specifically, `AEG` executes `iwconfig` using symbolic arguments (`argv`) as the input sources. `AEG` considers a variety of input sources, such as files, arguments, etc., by default.
2. After following the path `main` → `print_info` → `get_info`, `AEG` reaches line 15, where it detects an out-of-bounds memory error on variable

```
1  int main(int argc, char **argv) {
2    int skfd;              /* generic raw socket desc.      */
3    if(argc == 2)
4      print_info(skfd, argv[1], NULL, 0);
5  ...
6  static int print_info(int skfd, char *ifname, char *args[], int count)
       {
7    struct wireless_info  info;
8    int                   rc;
9    rc = get_info(skfd, ifname, &info);
10 ...
11 static int get_info(int skfd, char *ifname, struct wireless_info * info
       ) {
12   struct iwreq          wrq;
13   if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0) {
14       struct ifreq ifr;
15       strcpy(ifr.ifr_name, ifname); /* buffer overflow */
16 ...
```

Figure 1: Code snippet from Wireless Tools' `iwconfig`.

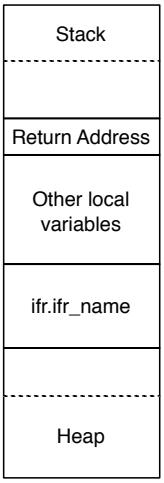

Figure 2: Memory Diagram

```
00000000  02 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000010  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000020  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000030  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00000040  01 01 01 01 70 f3 ff bf  31 c0 50 68 2f 2f 73 68  |....p...1.Ph//sh|
00000050  68 2f 62 69 6e 89 e3 50  53 89 e1 31 d2 b0 0b cd  |h/bin..PS..1....|
00000060  80 01 01 01 00                                    |.....|
```

Figure 3: A generated exploit of `iwconfig` from `AEG`.

`ifr.ifr_name`. `AEG` solves the current path constraints and generates a concrete input that will trigger the detected bug, e.g., the first argument has to be over 32 bytes.

3. `AEG` performs dynamic analysis on the `iwconfig` binary using the concrete input generated in step 2. It extracts runtime information about the memory layout, such as the address of the overflowed buffer (`ifr.ifr_name`) and the address of the return address of the vulnerable function (`get_info`).
4. `AEG` generates the constraints describing the exploit using the runtime information generated from the previous step: 1) the vulnerable buffer (`ifr.ifr_name`) must contain our shellcode, and 2) the overwritten return address must contain the address of the shellcode—available from runtime. Next, `AEG` appends the generated constraints to the path constraints and queries a constraint solver for a satisfying answer.
5. The satisfying answer gives us the exploit string, shown in Figure 3. Finally, `AEG` runs the program with the generated exploit and verifies that it works, i.e., spawns a shell. If the constraints were not solvable, `AEG` would resume searching the program for the next potential vulnerability.

Challenges. The above walkthrough illustrates a number of challenges that `AEG` has to address:

- The *State Space Explosion* problem (Steps 1-2). There are potentially an infinite number of paths that `AEG` has to explore until an exploitable path is detected. `AEG` utilizes preconditioned symbolic execution (see § 5.2) to target exploitable paths.
- The *Path Selection* problem (Steps 1-2). Amongst an infinite number of paths, `AEG` has to select which paths should be explored first. To do so, `AEG` uses path prioritization techniques (see § 5.3).
- The *Environment Modelling* problem (Steps 1-3). Real-world applications interact intensively with the underlying environment. To enable accurate analysis on such programs `AEG` has to model the environment IO behavior, including command-line arguments, files and network packets (see § 5.4).
- The *Mixed Analysis* challenge (Steps 1-4). `AEG` performs a mix of binary- and source-level analysis in order to scale to larger programs than could be handled with a binary-only approach. Combining the analyses' results of such fundamentally different levels of abstraction presents a challenge on its own (see § 6.2).
- The *Exploit Verification* problem (Step 5). Last,

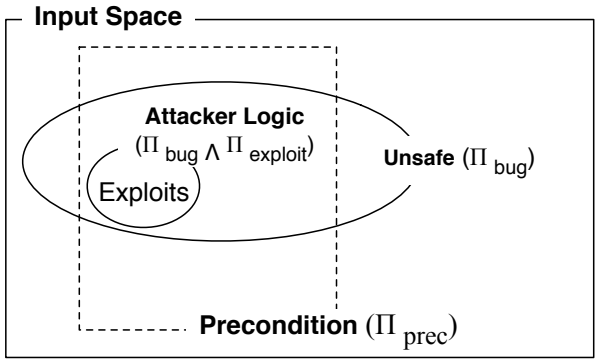

Figure 4: The input space diagram shows the relationship between unsafe inputs and exploits. Preconditioned symbolic execution narrows down the search space to inputs that satisfy the precondition (Π_{prec}).

`AEG` has to verify that the generated exploit is a *working* exploit for a given system (see § 6.3).

3 The AEG Challenge

At its core, the automatic exploit generation (AEG) challenge is a problem of finding program inputs that result in a desired exploited execution state. In this section, we show how the AEG challenge can be phrased as a formal verification problem, as well as propose a new symbolic execution technique that allows AEG to scale to larger programs than previous techniques. As a result, this formulation: 1) enables formal verification techniques to produce exploits, and 2) allows AEG to directly benefit from any advances in formal verification.

3.1 Problem Definition

In this paper we focus on generating a control flow hijack exploit input that intuitively accomplishes two things. First, the exploit should violate program safety, e.g., cause the program to write to out-of-bound memory. Second, the exploit must redirect control flow to the attacker's logic, e.g., by executing injected shellcode, performing a return-to-libc attack, and others.

At a high level, our approach uses program verification techniques where we verify that the program is exploitable (as opposed to traditional verification that verifies the program is safe). The exploited state is characterized by two Boolean predicates: a buggy execution path predicate Π_{bug} and a control flow hijack exploit predicate $\Pi_{exploit}$, specifying the control hijack and the code injection attack. The Π_{bug} predicate is satisfied when a program violates the semantics of program safety. However, simply violating safety is typically not enough. In addition, $\Pi_{exploit}$ captures the conditions needed to hijack control of the program.

An exploit in our approach is an input ε that satisfies the Boolean equation:

$$\Pi_{bug}(\varepsilon) \wedge \Pi_{exploit}(\varepsilon) = \texttt{true} \qquad (1)$$

Using this formulation, the mechanics of AEG is to check at each step of the execution whether Equation 1 is satisfiable. Any satisfying answer is, by construction, a control flow hijack exploit. We discuss these two predicates in more detail below.

The Unsafe Path Predicate Π_{bug}. Π_{bug} represents the path predicate of an execution that violates the safety property ϕ. In our implementation, we use popular well-known safety properties for C programs, such as checking for out-of-bounds writes, unsafe format strings, etc. The unsafe path predicate Π_{bug} partitions the input space into inputs that satisfy the predicate (unsafe), and inputs that do not (safe). While path predicates are sufficient to describe bugs at the source-code level, in AEG they are necessary but insufficient to describe the very specific actions we wish to take, e.g., execute shellcode.

The Exploit Predicate $\Pi_{exploit}$. The exploit predicate specifies the attacker's logic that the attacker wants to do after hijacking `eip`. For example, if the attacker only wants to crash the program, the predicate can be as simple as "set `eip` to an invalid address after we gain control". In our experiments, the attacker's goal is to get a shell. Therefore, $\Pi_{exploit}$ must specify that the shellcode is well-formed in memory, and that `eip` will transfer control to it. The conjunction of the exploit predicate ($\Pi_{exploit}$) will induce constraints on the final solution. If the final constraints (from Equation 1) are not met, we consider the bug as non-exploitable (§6.2).

3.2 Scaling with Preconditioned Symbolic Execution

Our formulation allows us to use formal verification techniques to generate exploits. While this means formal verification can be used for AEG, existing techniques such as model checking, weakest preconditions, and forward symbolic verification unfortunately only scale to small programs. For example, KLEE is a state-of-the-art forward symbolic execution engine [5], but in practice is limited to small programs such as `/bin/ls`. In our experiments, KLEE was able to find only 1 of the bugs we exploited (§ 8).

We observe that one reason scalability is limited with existing verification techniques is that they prove the absence of bugs by considering the entire program state space. For example, when KLEE explores a program for

buffer overflows it considers all possible input lengths up to some maximum size, i.e., inputs of length 0, inputs of length 1, and so on. We observe that we can scale AEG by restricting the state space to only include states that are likely exploitable, e.g., by considering only inputs of a minimum length needed to overwrite any buffer. We achieve this by performing low-cost analysis to determine the minimum length ahead of time, which allows us to prune off the state space search during the (more expensive) verification step.

We propose *preconditioned symbolic execution* as a verification technique for pruning off portions of the state space that are uninteresting. Preconditioned symbolic execution is similar to forward symbolic execution [16, 23] in that it incrementally explores the state space to find bugs. However, preconditioned symbolic execution takes in an additional Π_{prec} parameter. Preconditioned symbolic execution only descends into program branches that satisfy Π_{prec}, with the net effect that subsequent steps of unsatisfied branches are pruned away. [2] In AEG, we use preconditioned symbolic execution to restrict exploration to only likely-exploitable regions of the state space. For example, for buffer overflows Π_{prec} is specified via lightweight program analysis that determines the minimum sized input to overflow any buffer.

Figure 4 depicts the differences visually. Typical verification explores the entire input state space, as represented by the overall box, with the goal of finding inputs that are unsafe and satisfy Π_{bug}. In AEG, we are only concerned with the subset of unsafe states that are exploitable, represented by the circle labeled $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$. The intuition is that preconditioned symbolic execution limits the space searched to a smaller box.

Logically, we would be guaranteed to find all possible exploits when Π_{prec} is less restrictive than the exploitability condition:

$$\Pi_{\text{bug}}(x) \wedge \Pi_{\text{exploit}}(x) \Rightarrow \Pi_{\text{prec}}(x)$$

In practice, this restriction can be eased to narrow the search space even further, at the expense of possibly missing some exploits. We explore several possibilities in § 5.2, and empirically evaluate their effectiveness in § 8.

[2] Note preconditioned forward symbolic execution is different than weakest preconditions. Weakest preconditions statically calculate the weakest precondition to achieve a desired post-condition. Here we dynamically check a not-necessarily weakest precondition for pruning.

4 Our Approach

In this section, we give an overview of the components of AEG, our system for automatic exploit generation. Figure 5 shows the overall flow of generating an exploit in AEG. Our approach to the AEG challenge consists of six components: PRE-PROCESS, SRC-ANALYSIS, BUG-FIND, DBA [3], EXPLOIT-GEN, and VERIFY.

PRE-PROCESS: $\mathsf{src} \rightarrow (B_{\text{gcc}}, B_{\text{llvm}})$.
AEG is a two-input single-output system: the user provides the target binary and the LLVM bytecode of the same program, and—if AEG succeeds—we get back a working exploit for the given binary. Before the program analysis part begins, there is a necessary manual preprocessing step: the source program (*src*) is compiled down to 1) a binary B_{gcc}, for which AEG will try to generate a working exploit and 2) a LLVM bytecode file B_{llvm}, which will be used by our bug finding infrastructure.

SRC-ANALYSIS: $B_{\text{llvm}} \rightarrow max$.
AEG analyzes the source code to generate the maximum size of symbolic data *max* that should be provided to the program. AEG determines *max* by searching for the largest statically allocated buffers of the target program. AEG uses the heuristic that *max* should be at least 10% larger than the largest buffer size.

BUG-FIND $(B_{\text{llvm}}, \phi, max) \rightarrow (\Pi_{\text{bug}}, V)$.
BUG-FIND takes in LLVM bytecode B_{llvm} and a safety property ϕ, and outputs a tuple $\langle \Pi_{\text{bug}}, V \rangle$ for each detected vulnerability. Π_{bug} contains the path predicate, i.e., the conjunction of all path constraints up to the violation of the safety property ϕ. V contains source-level information about the detected vulnerability, such as the name of the object being overwritten, and the vulnerable function. To generate the path constraints, AEG uses a symbolic executor. The symbolic executor reports a bug to AEG whenever there is a violation of the ϕ property. AEG utilizes several novel bug-finding techniques to detect exploitable bugs (see § 5).

DBA: $(B_{\text{gcc}}, (\Pi_{\text{bug}}, V)) \rightarrow R$.
DBA performs dynamic binary analysis on the target binary B_{gcc} with a concrete buggy input and extracts runtime information R. The concrete input is generated by solving the path constraints Π_{bug}. While executing the vulnerable function (specified in V at the source-code level), DBA examines the binary to extract low-level runtime information (R),

[3] Dynamic Binary Analysis

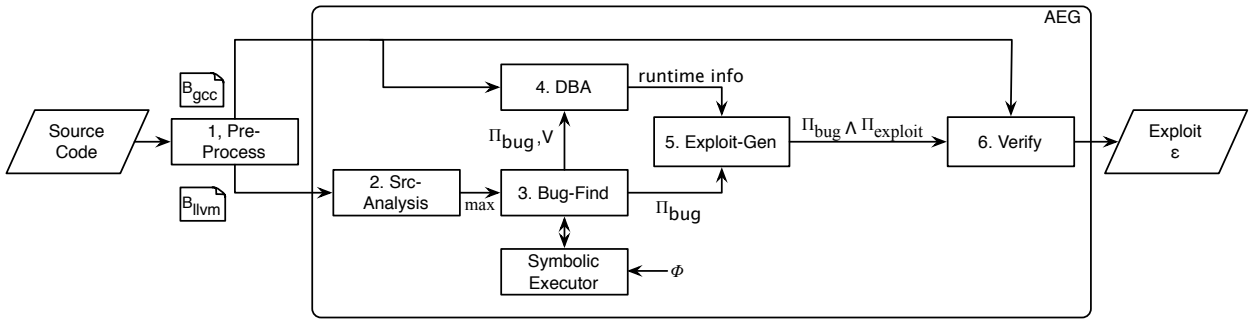

Figure 5: AEG design.

such as the vulnerable buffer's address on the stack, the address of the vulnerable function's return address, and the stack memory contents just before the vulnerability is triggered. DBA has to ensure that all the data gathered during this stage are accurate, since AEG relies on them to generate working exploits (see § 6.1).

EXPLOIT-GEN: $(\Pi_{\text{bug}}, R) \rightarrow \Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$.
EXPLOIT-GEN receives a tuple with the path predicate of the bug (Π_{bug}) and runtime information (R), and constructs a formula for a control flow hijack exploit. The output formula includes constraints ensuring that: 1) a possible program counter points to a user-determined location, and 2) the location contains shellcode (specifying the attacker's logic Π_{exploit}). The resulting exploit formula is the conjunction of the two predicates (see § 6.2).

VERIFY: $(B_{\text{gcc}}, \Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}) \rightarrow \{\varepsilon, \perp\}$.
VERIFY takes in the target binary executable B_{gcc} and an exploit formula $\Pi_{\text{bug}} \wedge \Pi_{\text{exploit}}$, and returns an exploit ε only if there is a satisfying answer. Otherwise, it returns $\perp$. In our implementation, AEG performs an additional step in VERIFY: runs the binary B_{gcc} with ε as an input, and checks if the adversarial goal is satisfied or not, i.e., if the program spawns a shell (see § 6.3).

Algorithm 1 shows our high-level algorithm for solving the AEG challenge.

Algorithm 1: Our AEG exploit generation algorithm

input : *src*: the program's source code
output: $\{\varepsilon, \perp\}$: a working exploit or $\perp$

```
(B_gcc, B_llvm) = Pre-Process(src);
max = Src-Analysis(B_llvm);
while (Π_bug, V) = Bug-Find(B_llvm, ϕ, max) do
    R = DBA(B_gcc, (Π_bug, V));
    Π_bug ∧ Π_exploit = Exploit-Gen(Π_bug, R);
    ε = Verify(B_gcc, Π_bug ∧ Π_exploit);
    if ε ≠ ⊥ then
        return ε;
return ⊥;
```

5 BUG-FIND: Program Analysis for Exploit Generation

BUG-FIND takes as input the target program in LLVM bytecode form, checks for bugs, and for each bug found attempts the remaining exploit generation steps until it succeeds. BUG-FIND finds bugs with symbolic program execution, which explores the program state space one path at a time. However, there are an infinite number of paths to potentially explore. AEG addresses this problem with two novel algorithms. First, we present a novel technique called preconditioned symbolic execution that constrains the paths considered to those that would most likely include exploitable bugs. Second, we propose novel path prioritization heuristics for choosing which paths to explore first with preconditioned symbolic execution.

5.1 Traditional Symbolic Execution for Bug Finding

At a high level, symbolic execution is conceptually similar to normal concrete execution except that we provide a fresh symbolic variable instead of providing a concrete value for inputs. As the program executes, each step of symbolic execution builds up an expression by substituting symbolic inputs for terms of the program. At program branches, the interpreter conceptually "forks off" two interpreters, adding the true branch guard to the conditions for the true branch interpreter, and similarly for the false branch. The conditions imposed as the interpreter executes are called the *path predicate* to exe-

cute the given path. After forking, the interpreter checks if the path predicate is satisfiable by querying a decision procedure. If not, the path is not realizable by any input, so the interpreter exits. If the path predicate can be satisfied, the interpreter continues executing and exploring the program state space. A more precise semantics can be found in Schwartz *et al.* [23].

Symbolic execution is used to find bugs by adding safety checks using ϕ. For example, whenever we access a buffer using a pointer, the interpreter needs to ensure the pointer is within the bounds of the buffer. The bounds-check returns either true, meaning the safety property holds, or false, meaning there is a violation, thus a bug. Whenever a safety violation is detected, symbolic execution stops and the current buggy path predicate (Π_{bug}) is reported.

5.2 Preconditioned Symbolic Execution

The main challenge with symbolic execution (and other verification techniques) is managing the state space explosion problem. Since symbolic execution forks off a new interpreter at every branch, the total number of interpreters is exponential in the number of branches.

We propose *preconditioned symbolic execution* as a novel method to target symbolic execution towards a certain subset of the input state space (shown in Figure 4). The state space subset is determined by the precondition predicate (Π_{prec}); inputs that do not satisfy Π_{prec} will not be explored. The intuition for preconditioned symbolic execution is that we can narrow down the state space we are exploring by specifying exploitability conditions as a precondition, e.g., all symbolic inputs should have the maximum size to trigger buffer overflow bugs. The main benefit from preconditioned symbolic execution is simple: by limiting the size of the input state space before symbolic execution begins, we can prune program paths and therefore explore the target program more efficiently.

Note that preconditions cannot be selected at random. If a precondition is too specific, we will detect no exploits (since exploitability will probably not imply the precondition); if it is too general, we will have to explore almost the entire state space. Thus, preconditions have to describe common characteristics among exploits (to capture as many as possible) and at the same time it should eliminate a significant portion of non-exploitable inputs.

Preconditioned symbolic execution enforces the precondition by adding the precondition constraints to the path predicate during initialization. Adding constraints may seem strange since there are more checks to perform at branch points during symbolic execution. However, the shrinking of the state space—imposed by the precondition constraints—outweighs the decision procedure overhead at branching points. When the precondition for a branch is unsatisfiable, we do no further checks and do not fork off an interpreter at all for the branch. We note that while we focus only on exploitable paths, the overall technique is more generally applicable.

```
1 int process_input(char input[42])
2     char buf[20];
3     while(input[i] != '\0')
4         buf[i++] = input[i];
```

Figure 6: Tight symbolic loops. A common pattern for most buffer overflows.

The advantages of preconditioned symbolic execution are best demonstrated via example. Consider the program shown in Figure 6. Suppose that the *input* buffer contains 42 symbolic bytes. Lines 4-5 represent a tight symbolic loop—equivalent to a `strcpy`—that will eventually spawn 42 different interpreters with traditional symbolic execution, each one having a different path predicate. The 1st interpreter will not execute the loop and will assume that $(input[0] = 0)$, the 2nd interpreter will execute the loop once and assume that $(input[0] \neq 0) \wedge (input[1] = 0)$, and so on. Thus, each path predicate will describe a different condition about the *string length* of the symbolic *input* buffer. [4]

Preconditioned symbolic execution avoids examining the loop iterations that will not lead to a buffer overflow by imposing a *length precondition*:

$$L = \forall_{i=0}^{i<n}(input[i] \neq 0) \wedge (input[n] = 0)$$

This predicate is appended to the path predicate (Π) before we start the symbolic execution of the program, thus eliminating paths that do not satisfy the precondition. In our previous example (Figure 6), the executor performs the followings checks every time we reach the loop branch point:

$$\begin{array}{ll} \text{false branch:} & \Pi \wedge L \Rightarrow input[i] = 0, \text{ pruned } \forall i < n \\ \text{true branch:} & \Pi \wedge L \Rightarrow input[i] \neq 0, \text{ satisfiable } \forall i < n \end{array}$$

Both checks are very fast to perform, since the validity (or invalidity) of the branch condition can be immediately determined by the precondition constraints L (in

[4] The length precondition for strings is generated based on a null character, because all strings are null-terminated.

fact, in this specific example there is no need for a solver query, since validity or invalidity can be determined by a simple iteration through our assumption set $\Pi \wedge L$). Thus, by applying the length precondition we only need a single interpreter to explore the entire loop. In the rest of the section, we show how we can generate different types of preconditions to reduce the search space.

5.2.1 Preconditions

In AEG, we have developed and implemented 4 different preconditions for efficient exploit generation:

None There is no precondition and the state space is explored as normal.

Known Length The precondition is that inputs are of known maximum length, as in the previous example. We use static analysis to automatically determine this precondition.

Known Prefix The precondition is that the symbolic inputs have a known prefix.

Concolic Execution Concolic execution [24] can be viewed as a specific form of preconditioned symbolic execution where the precondition is specified by a single program path as realized by an example input. For example, we may already have an input that crashes the program, and we use it as a precondition to determine if the executed path is exploitable.

The above preconditions assume varying amounts of static analysis or user input. In the following, we further discuss these preconditions, and also describe the reduction in the state space that preconditioned symbolic execution offers. A summary of the preconditions' effect on branching is shown in Figure 7.

None. Preconditioned symbolic execution is equivalent to standard symbolic execution. The input precondition is `true` (the entire state space). *Input Space*: For S symbolic input bytes, the size of the input space is 256^S. The example in Figure 7 contains $N+M$ symbolic branches and a symbolic loop with S maximum iterations, thus in the worst case (without pruning), we need $2^N \cdot S \cdot 2^M$ interpreters to explore the state space.

Known Length. The precondition is that all inputs should be of maximum length. For example, if the input data is of type string, we add the precondition that each byte of input up to the maximum input length is not NULL, i.e., (`strlen`($input$) = len) or equivalently in logic $(input[0] \neq 0) \wedge (input[1] \neq 0) \wedge \ldots \wedge (input[len-1] \neq 0) \wedge (input[len] = 0)$. *Input space*: The input space of a string of length len will be 255^{len}. Note that for $len = S$, this means a 0.4% decrease of the input space for each byte. *Savings*: The length precondition does not affect the $N+M$ symbolic branches of the example in Figure 7. However, the symbolic `strcpy` will be converted into a straight-line concrete copy — since we know the length and pruning is enabled, we need not consider copying strings of all possible lengths. Thus, we need 2^{N+M} interpreters to explore the entire state space. Overall, the length precondition decreases the input space slightly, but can concretize `strcpy`-like loops—a common pattern for detecting buffer overflows.

Known Prefix. The precondition constrains a prefix on input bytes, e.g., an HTTP GET request always starts with "GET", or that a specific header field needs to be within a certain range of values, e.g., the protocol field in the IP header. We use a prefix precondition to target our search towards inputs that start with that specific prefix. For example, suppose that we wish to explore only PNG images on an image-processing utility. The PNG standard specifies that all images must start with a standard 8-byte header PNG_H, thus simply by specifying a prefix precondition $(input[0] = PNG\_H[0]) \wedge \ldots \wedge (input[7] = PNG\_H[7])$, we can focus our search to PNG images alone. Note that prefix preconditions need not only consist of exact equalities; they can also specify a range or an enumeration of values for the symbolic bytes.

Input space: For S symbolic bytes and an exact prefix of length P ($P < N < S$), the size of the input space will be 256^{S-P}. *Savings*: For the example shown in Figure 7, the prefix precondition effectively concretizes the first P branches as well as the first P iterations of the symbolic `strcpy`, thus reducing the number of required interpreters to $S \cdot 2^{N+M-P}$. A prefix precondition can have a radical effect on the state space, but is no panacea. For example, by considering only valid prefixes we are potentially missing exploits caused by malformed headers.

Concolic Execution. The dual of specifying no precondition is specifying the precondition that all input bytes have a specific value. Specifying all input bytes have a specific value is equivalent to concolic execution [24]. Mathematically, we specify $\forall_i : \bigwedge (input[i] = concrete\_input[i])$.
Input Space: There is a single concrete input. *Savings*: A single interpreter is needed to explore the program, and because of state pruning, we are concretely executing the execution path for the given input. Thus, especially for concolic execution, it is much more useful to disable state pruning and drop the precondition constraints whenever we fork a new interpreter. Note that, in this case, AEG behaves as a concolic fuzzer, where

$$
\begin{array}{c}
N \\ \text{symbolic} \\ \text{branches}
\end{array}
\left\{
\begin{array}{ll}
if(\text{input[0]} < 42) & \ldots \\
\ldots & \\
if(\text{input[N-1]} < 42) & \ldots
\end{array}
\right.
$$

symbolic loop: strcpy(dest, input);

$$
\begin{array}{c}
M \\ \text{symbolic} \\ \text{branches}
\end{array}
\left\{
\begin{array}{ll}
if(\text{input[N]} < 42) & \ldots \\
if(\text{input[N+1]} < 42) & \ldots \\
\cdots & \\
if(\text{input[N+M-1]} < 42) & \ldots
\end{array}
\right.
$$

(a) An example that illustrates the advantages of preconditioned symbolic execution.

Precondition	Input Space	# of Interpreters
None	256^S	$2^N \cdot S \cdot 2^M$
Known Length	255^S	$2^N \cdot 2^M$
Known Prefix	256^{S-P}	$2^{N-P}(S-P)2^M$
Concolic	1	1

(b) The size of the input space and the number of interpreters required to explore the state space of the example program at the left, for each of the 4 preconditions supported by AEG. We use S to denote the number of symbolic input bytes and P for the length of the known prefix ($P < N < S$).

Figure 7: An example of preconditioned symbolic execution.

the concrete constraints describe the initial seed. Even though concolic execution seems to be the most constrained of all methods, it can be very useful in practice. For instance, an attacker may already have a proof-of-concept (PoC—an input that crashes the program) but cannot create a working exploit. AEG can take that PoC as a seed and generate an exploit—as long as the program is exploitable with any of the AEG-supported exploitation techniques.

5.3 Path Prioritization: Search Heuristics

Preconditioned symbolic execution limits the search space. However, within the search space, there is still the question of *path prioritization*: which paths should be explored first? AEG addresses this problem with path-ranking heuristics. All pending paths are inserted into a priority queue based on their ranking, and the next path to explore is always drawn out of the priority queue. In this section, we present two new path prioritization heuristics we have developed: *buggy-path-first* and *loop exhaustion*.

Buggy-Path-First. Exploitable bugs are often preceded by small but unexploitable mistakes. For example, in our experiments we found errors where a program first has an off-by-one error in the amount of memory allocated for a `strcpy`. While the off-by-one error could not directly be exploited, it demonstrated that the programmer did not have a good grasp of buffer bounds. Eventually, the length misunderstanding was used in another statement further down the path that was exploitable. The observation that one bug on a path means subsequent statements are also likely to be buggy (and hopefully exploitable) led us to the buggy-path-first heuristic. Instead of simply reporting the first bug and stopping like other tools such as KLEE [5], the buggy-path-first heuristic prioritizes buggy paths higher and continues exploration.

Loop Exhaustion. Loops whose exit condition depends on symbolic input may spawn a tremendous amount of interpreters—even when using preconditioned symbolic execution techniques such as specifying a maximum length. Most symbolic execution approaches mitigate this program by de-prioritizing subsequent loop executions or only considering loops a small finite number of times, e.g., up to 3 iterations. While traditional loop-handling strategies are excellent when the main goal is maximizing code coverage, they often miss exploitable states. For example, the perennial exploitable bug is a `strcpy` buffer overflow, where the `strcpy` is essentially a while loop that executes as long as the source buffer is not NULL. Typical buffer sizes are quite large, e.g., 512 bytes, which means we must execute the loops at least that many times to create an exploit. Traditional approaches that limit loops simply miss these bugs.

We propose and use a *loop exhaustion* search strategy. The loop-exhaustion strategy gives higher priority to an interpreter exploring the maximum number of loop iterations, hoping that computations involving more iterations are more promising to produce bugs like buffer overflows. Thus, whenever execution hits a symbolic loop, we try to *exhaust* the loop—execute it as many times as possible. Exhausting a symbolic loop has two immediate side effects: 1) on each loop iteration a new interpreter is spawned, effectively causing an explosion in the state space, and 2) execution might get "stuck"

in a deep loop. To avoid getting stuck, we impose two additional heuristics during loop exhaustion: 1) we use preconditioned symbolic execution along with pruning to reduce the number of interpreters or 2) we give higher priority to only one interpreter that tries to fully exhaust the loop, while all other interpreters exploring the same loop have the lowest possible priority.

5.4 Environment Modelling: Vulnerability Detection in the Real World

AEG models most of the system environments that an attacker can possibly use as an input source. Therefore, AEG can detect most security relevant bugs in real programs. Our support for environment modeling includes file systems, network sockets, standard input, program arguments, and environment variables. Additionally, AEG handles most common system and library function calls.

Symbolic Files. AEG employs an approach similar to KLEE's [5] for symbolic files: modeling the fundamental system call functions, such as open, read, and write. AEG simplifies KLEE's file system models to speedup the analysis, since our main focus is not on code coverage, but on efficient exploitable bug detection. For example, AEG ignores symbolic file properties such as permissions, in order to avoid producing additional paths.

Symbolic Sockets. To be able to produce remote exploits, AEG provides network support in order to analyze networking code. A symbolic socket descriptor is handled similarly to a symbolic file descriptor, and symbolic network packets and their payloads are handled similarly to symbolic files and their contents. AEG currently handles all network-related functions, including `socket`, `bind`, `accept`, `send`, etc.

Environment Variables. Several vulnerabilities are triggered because of specific environment variables. Thus, AEG supports a complete summary of `get_env`, representing all possible results (concrete values, fully symbolic and failures).

Library Function Calls and System Calls. AEG provides support for about 70 system calls. AEG supports all the basic network system calls, thread-related system calls, such as `fork`, and also all common formatting functions, including `printf` and `syslog`. Threads are handled in the standard way, i.e., we spawn a new symbolic interpreter for each process/thread creation function invocation. In addition, AEG reports a possibly exploitable bug whenever a (fully or partially) symbolic argument is passed to a formatting function. For instance, AEG will detect a format string vulnerability for "`fprintf`(stdout, user_input)".

6 DBA, EXPLOIT-GEN and VERIFY: The Exploit Generation

At a high level, the three components of AEG (DBA, EXPLOIT-GEN and VERIFY) work together to convert the unsafe predicate (Π_{bug}) output by BUG-FIND into a working exploit ε.

6.1 DBA: Dynamic Binary Analysis

DBA is a dynamic binary analysis (instrumentation) step. It takes in three inputs: 1) the target executable (B_{gcc}) that we want to exploit; 2) the path constraints that lead up to the bug (Π_{bug}); and 3) the names of vulnerable functions and buffers, such as the buffer susceptible to overflow in a stack overflow attack or the buffer that holds the malicious format string in a format string attack. It then outputs a set of runtime information: 1) the address to overwrite (in our implementation, this is the address of the return address of a function, but we can easily extend this to include function pointers or entries in the GOT), 2) the starting address that we write to, and 3) the additional constraints that describe the stack memory contents just before the bug is triggered.

Once AEG finds a bug, it replays the same buggy execution path using a concrete input. The concrete input is generated by solving the path constraints Π_{bug}. During DBA, AEG performs instrumentation on the given executable binary B_{gcc}. When it detects the vulnerable function call, it stops execution and examines the stack. In particular, AEG obtains the address of the return address of the vulnerable function (&retaddr), the address of the vulnerable buffer where the overwrite starts (bufaddr) and the stack memory contents between them (μ).

In the case of format string vulnerabilities, the vulnerable function is a *variadic* formatting function that takes user input as the format argument. Thus, the address of the return address (&retaddr) becomes the return address of the vulnerable formatting function. For example, if there is a vulnerable `printf` function in a program, AEG overwrites the return address of the `printf` function itself, exploiting the format string vulnerability. This way, an attacker can hijack control of the program right after the vulnerable function returns. It is straightforward to adapt additional format string attacks such as GOT hijacking, in AEG.

Stack Restoration. AEG examines the stack contents during DBA in order to generate an exploit predicate ($\Pi_{bug} \wedge \Pi_{exploit}$) that does not corrupt the local stack variables in EXPLOIT-GEN (§ 6.2). For example, if there is a dereference from the stack before the vulner-

```
1  char *ptr = malloc(100);
2  char buf[100];
3  strcpy(buf, input); // overflow
4  strcpy(ptr, buf);   // ptr dereference
5  return;
```

Figure 8: When stack contents are garbled by stack overflow, a program can fail before the return instruction.

able function returns, simply overwriting the stack will not always produce a valid exploit. Suppose an attacker tries to exploit the program shown in Figure 8 using the `strcpy` buffer overflow vulnerability. In this case, `ptr` is located between the return address and the `buf` buffer. Note that `ptr` is dereferenced after the stack overflow attack. Since `ptr` is also on the stack, the contents of `ptr` are garbled by the stack overflow, and might cause the program to crash before the return instruction. Thus, a sophisticated attack must consider the above case by overwriting a valid memory pointer to the stack. AEG properly handles this situation by examining the entire stack space during DBA, and passing the information (μ) to EXPLOIT-GEN.

6.2 Exploit-Gen

EXPLOIT-GEN takes in two inputs to produce an exploit: the unsafe program state containing the path constraints (Π_{bug}) and low-level runtime information R, i.e., the vulnerable buffer's address (bufaddr), the address of the vulnerable function's return address (&retaddr), and the runtime stack memory contents (μ). Using that information, EXPLOIT-GEN generates exploit formulas ($\Pi_{bug} \wedge \Pi_{exploit}$) for four types of exploits: 1) stack-overflow return-to-stack, 2) stack-overflow return-to-libc, 3) format-string return-to-stack, 4) format-string return-to-libc. In this paper, we present the full algorithm only for 1. The full algorithms for the rest of our exploitation techniques can be found on our website [2].

In order to generate exploits, AEG performs two major steps. First, AEG determines the class of attack to perform and formulates $\Pi_{exploit}$ for control hijack. For example, in a stack-overflow return-to-stack attack, $\Pi_{exploit}$ must have the constraint that the address of the return address (&retaddr) should be overwritten to contain the address of the shellcode—as provided by DBA. Further, the exploit predicate $\Pi_{exploit}$ must also contain constraints that shellcode must be written on the target buffer. The generated predicate is used in conjunction with Π_{bug} to produce the final constraints (the exploit formula $\Pi_{bug} \wedge \Pi_{exploit}$) that can be solved to produce

Algorithm 2: Stack-Overflow Return-to-Stack Exploit Predicate Generation Algorithm

input : (bufaddr, &retaddr, μ) = R
output: $\Pi_{exploit}$

1 **for** $i = 1$ *to len*(μ) **do**
2 | $exp\_str[i] \leftarrow \mu[i]$; `// stack restoration`
3 offset ← &retaddr - bufaddr;
4 jmp_target ← offset + 8 ; `// old ebp + retaddr = 8`
5 exp_str[offset] ← jmp_target ; `// eip hijack`
6 **for** $i = 1$ *to len(shellcode)* **do**
7 | exp_str[offset + i] ← shellcode[i];
8 **return** $(Mem[bufaddr] == exp\_str[1]) \wedge \ldots \wedge (Mem[bufaddr + len(\mu) - 1] == exp\_str[len(\mu)])$; `//` $\Pi_{exploit}$

an exploit. Algorithm 2 shows how the exploit predicate ($\Pi_{exploit}$) is generated for stack-overflow return-to-stack attacks.

6.2.1 Exploits

AEG produces two types of exploits: return-to-stack [21] and return-to-libc [10], both of which are the most popular classic control hijack attack techniques. AEG currently cannot handle state-of-the-art protection schemes, but we discuss possible directions in § 9. Additionally, our return-to-libc attack is different from the classic one in that we do not need to know the address of a "/bin/sh" string in the binary. This technique allows bypassing stack randomization (but not libc randomization).

Return-to-stack Exploit. The return-to-stack exploit overwrites the return address of a function so that the program counter points back to the injected input, e.g., user-provided shellcode. To generate the exploit, AEG finds the address of the vulnerable buffer (bufaddr) into which an input string can be copied, and the address where the return address of a vulnerable function is located at. Using the two addresses, AEG calculates the jump target address where the shellcode is located. Algorithm 2 describes how to generate an exploit predicate for a stack overflow vulnerability in the case of a return-to-stack exploit where the shellcode is placed after the return address.

Return-to-libc Exploit. In the classic return-to-libc attack, an attacker usually changes the return address to point to the `execve` function in libc. However, to spawn a shell, the attacker must know the address of a "/bin/sh" string in the binary, which is not common in most programs. In our return-to-libc attack, we create a symbolic link to `/bin/sh` and for the link name we use an arbitrary string which resides in libc. For exam-

ple, a 5 byte string pattern $e8..00....._{16}$ [5] is very common in libc, because it represents a call instruction on x86. Thus, AEG finds a certain string pattern in libc, and generates a symbolic link to /bin/sh in the same directory as the target program. The address of the string is passed as the first argument of execve (the file to execute), and the address of a null string 00000000_{16} is used for the second and third arguments. The attack is valid only for local attack scenarios, but is more reliable since it bypasses stack address randomization.

Note that the above exploitation techniques (return-to-stack and return-to-libc) determine how to spawn a shell for a control hijack attack, but not how to hijack the control flow. Thus, the above techniques can be applied by different types of control hijack attacks, e.g., format string attacks and stack overflows. For instance, a format string attack can use either of the above techniques to spawn a shell. AEG currently handles all possible combinations of the above attack-exploit patterns.

6.2.2 Exploitation Techniques

Various Shellcode. The return-to-stack exploit requires shellcode to be injected on the stack. To support different types of exploits, AEG has a shellcode database with two shellcode classes: standard shellcodes for local exploits, and binding and reverse binding shellcodes for remote exploits. In addition, this attack restores the stack contents by using the runtime information μ (§ 6.1).

Types of Exploits. AEG currently supports four types of exploits: stack-overflow return-to-stack, stack-overflow return-to-libc, format-string return-to-stack, and format-string return-to-libc exploit. The algorithms to generate the *exp_str* for each of the above exploits are simple extensions of Algorithm 2. The interested reader may refer to our website [2] for the full algorithms.

Shellcode Format & Positioning. In code-injection attack scenarios, there are two parameters that we must always consider: 1) the format, e.g., size and allowed characters and 2) the positioning of the injected shellcode. Both are important because advanced attacks have complex requirements on the injected payload, e.g., that the exploit string fits within a limited number of bytes or that it only contains alphanumeric characters. To find positioning, AEG applies a brute-force approach: tries every possible user-controlled memory location to place the shellcode. For example, AEG can place the shellcode either below or above the overwritten return address. To address the special formatting challenge, AEG has a shellcode database containing about 20 different shellcodes, including standard and alphanumeric. Again, AEG tries all possible shellcodes in order to increase reliability. Since AEG has a VERIFY step, all the generated control hijacks are verified to become actual exploits.

6.2.3 Reliability of Exploits

Exploits are delicate, especially those that perform control flow hijacking. Even a small change, e.g., the way a program executes either via ./a.out or via ../../../a.out, will result in a different memory layout of the process. This problem persists even when ASLR is turned off. For the same reason, most of the proof-of-concept exploits in popular advisories do not work in practice without some (minor or major) modification. In this subsection, we discuss the techniques employed by AEG to generate reliable exploits for a given system configuration: a) offsetting the difference in environment variables, and b) using NOP-sleds.

Offsetting the Difference in Environment Variables. Environment variables are different for different terminals, program arguments of different length, etc. When a program is first loaded, environment variables will be copied onto the program's stack. Since the stack grows towards lower memory addresses, the more environment variables there are, the lower the addresses of the actual program data on the stack are going to be. Environment variables such as OLDPWD and _ (underscore) change even across same system, since the way the program is invoked matters. Furthermore, the arguments (argv) are also copied onto the stack. Thus, the length of the command line arguments affects the memory layout. Thus, AEG calculates the addresses of local variables on the stack based upon the difference in the size of the environment variables between the binary analysis and the normal run. This technique is commonly used if we have to craft the exploit on a machine and execute the exploit on another machine.

NOP-Sled. AEG optionally uses NOP-sleds. For simplicity, Algorithm 2 does not take the NOP-sled option into account. In general, a large NOP-sled can make an exploit more reliable, especially against ASLR protection. On the other hand, the NOP-sled increases the size of the payload, potentially making the exploit more difficult or impossible. In AEG's case, the NOP-sled option can be either turned on or off by a command line option.

6.3 Verify

VERIFY takes in two inputs: 1) the exploit constraints $\Pi_{bug} \wedge \Pi_{exploit}$, and 2) the target binary. It outputs either

[5] A dot (.) represents a 4-bit string in hexadecimal notation.

a concrete working exploit, i.e., an exploit that spawns a shell, or $\perp$, if AEG fails to generate the exploit. VERIFY first solves the exploit constraints to get a concrete exploit. If the exploit is a local attack, it runs the executable with the exploit as the input and checks if a shell has been spawned. If the exploit is a remote attack, AEG spawns three processes. The first process runs the executable. The second process runs nc to send the exploit to the executable. The third process checks that a remote shell has been spawned at port 31337.

Note that, in Figure 5, we have shown a straight-line flow from PRE-PROCESS to VERIFY for simplicity. However, in the actual system, VERIFY provides feedback to EXPLOIT-GEN if the constraints cannot be solved. This is a cue for EXPLOIT-GEN to select a different shellcode.

7 Implementation

AEG is written in a mixture of C++ and Python and consists of 4 major components: symbolic executor (BUG-FIND), dynamic binary evaluator (DBA), exploit generator (EXPLOIT-GEN), and constraint solver (VERIFY). We chose KLEE [5] as our backend symbolic executor, and added about 5000 lines of code to implement our techniques and heuristics as well as to add in support for other input sources (such as sockets and symbolic environment variables). Our dynamic binary evaluator was written in Python, using a wrapper for the GNU debugger [22]. We used STP for constraint solving [12].

8 Evaluation

The following sections present our experimental work on the AEG challenge. We first describe the environment in which we conducted our experiments. Then, we show the effectiveness of AEG by presenting 16 exploits generated by AEG for 14 real-world applications. Next, we highlight the importance of our search heuristics—including preconditioned symbolic execution—in identifying exploitable bugs. In addition, we present several examples illustrating the exploitation techniques already implemented in AEG. Last, we evaluate the reliability of the generated exploits. For a complete explanation of each generated exploit and more experimental results, we refer the reader to our website [2].

8.1 Experimental Setup

We evaluated our algorithms and AEG on a machine with a 2.4 GHz Intel(R) Core 2 Duo CPU and 4GB of RAM with 4MB L2 Cache. All experiments were performed under Debian Linux 2.6.26-2. We used LLVM-GCC 2.7 to compile programs to run in our source-based AEG and GCC 4.2.4 to build binary executables. All programs presented in the paper are unmodified open-source applications that people use and can be downloaded from the Internet. Time measurements are performed with the Unix time command. The buggy-path-first and loop exhaustion search heuristics elaborated in § 5.3 were turned on by default for all the experiments.

8.2 Exploits by AEG

Table 1 shows the list of vulnerabilities that AEG successfully exploits. We found these 14 programs from a variety of popular advisories: Common Vulnerabilities and Exposures (CVE), Open Source Vulnerability Database (OSVDB), and Exploit-DB (EDB) and downloaded them to test on AEG. Not only did AEG reproduce the exploits provided in the CVEs, it found and generated working exploits for 2 additional vulnerabilities — 1 for expect-5.43 and 1 for htget-0.93.

We order the table by the kind of path exploration technique used to find the bug, ordered from the least to most amount of information given to the algorithm itself. 4 exploits required no precondition at all and paths were explored using only our path prioritization techniques (§ 5.3). We note that although we build on top of KLEE [5], in our experiments KLEE only detected the iwconfig exploitable bug.

6 of the exploits were generated only after inferring the possible maximum lengths of symbolic inputs using our static analysis (the Length rows). Without the maximum input length AEG failed most often because symbolic execution would end up considering all possible input lengths up to some maximum buffer size, which was usually very large (e.g., 512 bytes). Since length is automatically inferred, these 6 combined with the previous 4 mean that 10 total exploits were produced automatically with no additional user information.

5 exploits required that the user specify a prefix on the input space to explore. For example, xmail's vulnerable program path is only triggered with valid a email address. Therefore, we needed to specify to AEG that the input included an "@" sign to trigger the vulnerable path.

Corehttp is the only vulnerability that required concolic execution. The input we provided was "A"x (repeats 880 times) + \r\n\r\n. Without specifying the complete GET request, symbolic execution got stuck on exploring where to place white-spaces and EOL (end-of-line) characters.

Generation Time. Column 5 in Table 1 shows the total time to generate working exploits. The quickest we generated an exploit was 0.5*s* for iwconfig (with a length

	Program	Ver.	Exploit Type	Vulnerable Input src	Gen. Time (sec.)	Executable Lines of Code	Advisory ID.
None	aeon	0.2a	Local Stack	Env. Var.	3.8	3392	CVE-2005-1019
	iwconfig	V.26	Local Stack	Arguments	1.5	11314	CVE-2003-0947
	glftpd	1.24	Local Stack	Arguments	2.3	6893	OSVDB-ID#16373
	ncompress	4.2.4	Local Stack	Arguments	12.3	3198	CVE-2001-1413
Length	htget (processURL)	0.93	Local Stack	Arguments	57.2	3832	CVE-2004-0852
	htget (HOME)	0.93	Local Stack	Env. Var	1.2	3832	Zero-day
	expect (DOTDIR)	5.43	Local Stack	Env. Var	187.6	458404	Zero-day
	expect (HOME)	5.43	Local Stack	Env. Var	186.7	458404	OSVDB-ID#60979
	socat	1.4	Local Format	Arguments	3.2	35799	CVE-2004-1484
	tipxd	1.1.1	Local Format	Arguments	1.5	7244	OSVDB-ID#12346
Prefix	aspell	0.50.5	Local Stack	Local File	15.2	550	CVE-2004-0548
	exim	4.41	Local Stack	Arguments	33.8	241856	EDB-ID#796
	xserver	0.1a	Remote Stack	Sockets	31.9	1077	CVE-2007-3957
	rsync	2.5.7	Local Stack	Env. Var	19.7	67744	CVE-2004-2093
	xmail	1.21	Local Stack	Local File	1276.0	1766	CVE-2005-2943
Concolic	corehttp	0.5.3	Remote Stack	Sockets	83.6	4873	CVE-2007-4060
Average Generation Time & Executable Lines of Code					114.6	56784	

Table 1: List of open-source programs successfully exploited by `AEG`. Generation time was measured with the GNU Linux `time` command. Executable lines of code was measured by counting LLVM instructions.

precondition), which required exploring a single path. The longest was xmail at 1276*s* (a little over 21 minutes), and required exploring the most paths. On average exploit generation took 114.6*s* for our test suite. Thus, when AEG works, it tends to be very fast.

Variety of Environment Modeling. Recall from § 5.4, AEG handles a large variety of input sources including files, network packets, etc. In order to present the effectiveness of AEG in environment modeling, we grouped the examples by exploit type (Table 1 column 4), which is either local stack (for a local stack overflow), local format (for a local format string attack) or remote stack (for a remote stack overflow) and input source (column 5), which shows the source where we provide the exploit string. Possible sources of user input are environment variables, network sockets, files, command line arguments and `stdin`.

The two zero-day exploits, expect and htget, are both environment variable exploits. While most attack scenarios for environment variable vulnerabilities such as these are not terribly exciting, the main point is that AEG found new vulnerabilities and exploited them automatically.

8.3 Preconditioned Symbolic Execution and Path Prioritization Heuristics

8.3.1 Preconditioned Symbolic Execution

We also performed experiments to show how well preconditioned symbolic execution performs on specific vulnerabilities when different preconditions are used. Figure 9 shows the result. We set the maximum analysis time to 10,000 seconds, after which we terminate the program. The preconditioned techniques that failed to detect an exploitable bug within the time limit are shown as a bar of maximum length in Figure 9.

Our experiments show that increasing the amount of information supplied to the symbolic executor via the precondition significantly improves bug detection times

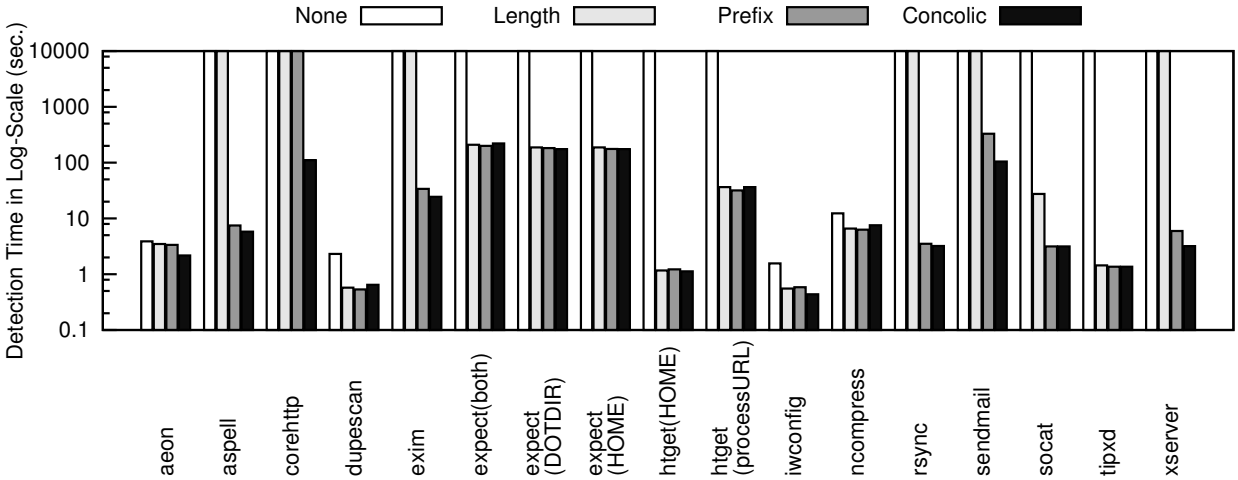

Figure 9: Comparison of preconditioned symbolic execution techniques.

and thus the effectiveness of AEG. For example, by providing a length precondition we almost tripled the number of exploitable bugs that AEG could detect within the time limit. However, the amount of information supplied did not tremendously change how quickly an exploit is generated, when it succeeds at all.

8.3.2 Buggy-Path-First: Consecutive Bug Detection

Recall from § 5.3 the path prioritization heuristic to check buggy paths first. `tipxd` and `htget` are example applications where this prioritization heuristic pays off. In both cases there is a non-exploitable bug followed by an exploitable bug in the same path. Figure 10 shows a snippet from `tipxd`, where there is an initial non-exploitable bug on line 1 (it should be "`malloc(strlen(optarg) + 1)`" for the NULL byte). AEG recognizes that the bug is non-exploitable and prioritizes that path higher for continued exploration.

Later on the path, AEG detects a format string vulnerability on line 10. Since the `config_filename` is set from the command line argument `optarg` in line 5, we can pass an arbitrary format string to the `syslog` function in line 10 via the variable `log_entry`. AEG recognizes the format string vulnerability and generates a format string attack by crafting a suitable command line argument.

8.4 Mixed Binary and Source Analysis

In § 1, we argue that source code analysis alone is insufficient for exploit generation because low-level runtime details like stack layout matter. The `aspell, htget, corehttp, xserver` are examples of this axiom.

```
1 int ProcessURL(char *TheURL, char *
      Hostname, char *Filename, char *
      ActualFilename, unsigned *Port) {
2   char BufferURL[MAXLEN];
3   char NormalURL[MAXLEN];
4   strcpy(BufferURL, TheURL);
5   ...
6   strncpy(Hostname, NormalURL, I);
```

Figure 11: Code snippet of `htget`

For example, Figure 11 shows a code snippet from `htget`. The stack frame when invoking this function has the function arguments at the top of the stack, then the return address and saved `ebp`, followed by the local buffers `BufferURL` and `NormalURL`. The `strcpy` on line 4 is exploitable where `TheURL` can be much longer than `BufferURL`. However, we must be careful in the exploit to *only* overwrite up to the return address, e.g., if we overwrite the return address and `Hostname`, the program will simply crash when `Hostname` is dereferenced (before returning) on line 6.

Since our technique performs dynamic analysis, we can reason about runtime details such as the exact stack layout, exactly how many bytes the compiler allocated to a buffer, etc, very precisely. For the above programs this precision is essential, e.g., in `htget` the predicate asserts that we overwrite up to the return address but no further. If there is not enough space to place the payload before the return address, AEG can still generate an ex-

```
1  if(!(sysinfo.config_filename = malloc(strlen(optarg)))) {
2      fprintf(stderr, "Could not allocate memory for filename storage\n");
3      exit(1);
4  }
5  strcpy((char *)sysinfo.config_filename, optarg);
6  tipxd_log(LOG_INFO, "Config file is %s\n", sysinfo.config_filename);
7  ...
8  void tipxd_log(int priority, char *format, ...) {
9      vsnprintf(log_entry, LOG_ENTRY_SIZE-1, format, ap);
10     syslog(priority, log_entry);
```

Figure 10: Code snippet of `tipxd`.

ploit by applying stack restoration (presented in § 6.1), where the local variables and function arguments are overwritten, but we impose constraints that their values should remain unchanged. To do so, AEG again relies on our dynamic analysis component to retrieve the runtime values of the local variables and arguments.

8.5 Exploit Variants

Whenever an exploitable bug is found, AEG generates an exploit formula ($\Pi_{bug} \wedge \Pi_{exploit}$) and produces an exploit by finding a satisfying answer. However, this does not mean that there is a *single* satisfying answer (exploit). In fact, we expected that there is huge number of inputs that satisfy the formula. To verify our expectations, we performed an additional experiment where we configured AEG to generate *exploit variants*—different exploits produced by the same exploit formula. Table 2 shows the number of exploit variants generated by AEG within an hour for 5 sample programs.

Program	# of exploits
iwconfig	3265
ncompress	576
aeon	612
htget	939
glftpd	2201

Table 2: Number of exploit variants generated by AEG within an hour.

8.6 Additional Success

AEG also had an anecdotal success. Our research group entered smpCTF 2010 [27], a time-limited international competition where teams compete against each other by solving security challenges. One of the challenges was to exploit a given binary. Our team ran the Hex-rays decompiler to produce source, which was then fed into AEG (with a few tweaks to fix some incorrect decompilation from the Hex-rays tool). AEG returned an exploit in under 60 seconds.

9 Discussion and Future Work

Advanced Exploits. In our experiments we focused on stack buffer overflows and format string vulnerabilities. In order to extend AEG to handle heap-based overflows we would likely need to extend the control flow reasoning to also consider heap management structures. Integer overflows are more complicated however, as typically an integer overflow is not problematic by itself. Security-critical problems usually appear when the overflowed integer is used to index or allocate memory. We leave adding support for these types of vulnerabilities as future work.

Other Exploit Classes. While our definition includes the most popular bugs exploited today, e.g., input validation bugs, such as information disclosure, buffer overflows, heap overflows, and so on, it does not capture all security-critical vulnerabilities. For example, our formulation leaves out-of-scope timing attacks against crypto, which are not readily characterized as safety problems. We leave extending AEG to these types of vulnerabilities as future work.

Symbolic Input Size. Our current approach performs simple static analysis and determines that symbolic input variables should be 10% larger in size than the largest statically allocated buffer. While this is an improvement over KLEE (KLEE required a user specify the size), and was sufficient for our examples, it is somewhat simplistic. More sophisticated analysis would provide greater precision for exactly what may be exploitable, e.g., by considering stack layout, and may be necessary for more advanced exploits, e.g., heap overflows where buffers are dynamically allocated.

Portable Exploits. In our approach, AEG produces an exploit for a given environment, i.e., OS, compiler, etc. For example, if AEG generates an exploit for a GNU compiled binary, the same exploit might not work for a binary compiled with the Intel compiler. This is to be expected since exploits are dependent upon run-time layout that may change from compiler to compiler. However, given an exploit that works when compiled with A, we can run AEG on the binary produced from compiler B to check if we can create a new exploit. Also, our current prototype only handles Linux-compatible exploits. Crafting platform-independent and portable exploits is addressed in other work [7] and falls outside the scope of this paper.

10 Related Work

Automatic Exploit Generation. Brumley *et al.* [4] introduced the automatic *patch-based* exploit generation (APEG) challenge. They also introduced the notion that exploits can be described as a predicate on the program state space, which we use and refine in this work. There are two significant differences between AEG and APEG. First, APEG requires access to a buggy program and a patch, while AEG only requires access to a potentially buggy program. Second, APEG defines an exploit as an input violating a new safety check introduced by a patch, e.g., only generating unsafe inputs in Figure 4. While Brumley *et al.* speculate generating root shells may be possible, they do not demonstrate it. We extend their notion of "exploit" to include specific actions, and demonstrate that we can produce specific actions such as launch a shell. Previously, Heelan *et al.* [13] automatically generated a control flow hijack when the bug is known, given a crashing input (similar to concolic execution), and a trampoline register is known.

Bug-finding techniques. In blackbox fuzzing, we give random inputs to a program until it fails or crashes [19]. Blackbox fuzzing is easy and cheap to use, but it is hard to use in a complex program. Symbolic execution has been used extensively in several application domains, including vulnerability discovery and test case generation [5, 6], input filter generation [3, 8], and others. Symbolic execution is so popular because of its simplicity: it behaves just like regular execution but it also allows data (commonly input) to be symbolic. By performing computations on symbolic data instead of their concrete values, symbolic execution allows us to reason about multiple inputs with a single execution. Taint analysis is a type of information flow analysis for determining whether untrusted user input can flow into trusted sinks. There are both static [15, 18, 26] and dynamic [20, 28] taint analysis tools. For a more extensive explanation of symbolic execution and taint analysis, we refer to a recent survey [23].

Symbolic Execution There is a rich variety of work in symbolic execution and formal methods that can be applied to our AEG setting. For example, Engler *et al.* [11] mentioned the idea of *exactly-constrained* symbolic execution, where equality constraints are imposed on symbolic data for concretization, and Jager *et al.* introduce directionless weakest preconditions that can produce the formulas needed for exploit generation potentially more efficiently [14]. Our problem definition enables any form of formal verification to be used, thus we believe working on formal verification is a good place to start when improving AEG.

11 Conclusion

In this paper, we introduced the first fully automatic end-to-end approach for exploit generation. We implemented our approach in AEG and analyzed 14 open-source projects. We successfully generated 16 control-flow hijack exploits, two of which were against previously unknown vulnerabilities. In order to make AEG practical, we developed a novel preconditioned symbolic execution technique and path prioritization algorithms for finding and identifying exploitable bugs.

12 Acknowledgements

We would like to thank all the people that worked in the AEG project and especially JongHyup Lee, David Kohlbrenner and Lokesh Agarwal. We would also like to thank our anonymous reviewers for their useful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 0953751. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work is also partially supported by grants from Northrop Grumman as part of the Cybersecurity Research Consortium, from Lockheed Martin, and from DARPA Grant No. N10AP20021.

References

[1] AEG. automatic exploit generation demo. http://www.youtube.com/watch?v=M_nuEDT-xaw, Aug. 2010.

[2] D. Brumley. http://security.ece.cmu.edu.

[3] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *IEEE Transactions on*

Dependable and Secure Computing, 5(4):224–241, Oct. 2008.

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

[5] 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 Symposium on Operating System Design and Implementation*, 2008.

[6] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2006.

[7] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton. Platform-independent programs. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the ACM Symposium on Operating System Principles*, Oct. 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating System Principles*, 2005.

[10] S. Designer. "return-to-libc" attack. Bugtraq, Aug. 1997.

[11] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *International Symposium on Software Testing and Analysis*, pages 1–4, 2007.

[12] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings on the Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, July 2007.

[13] S. Heelan. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Technical Report MSc Thesis, Oxford University, 2002.

[14] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, Feb. 2010.

[15] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the USENIX Security Symposium*, 2004.

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

[17] C. Lattner. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Symposium on Code Generation and Optimization*, 2004.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

[19] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

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

[21] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996. File 14/16.

[22] PyGDB. Python wrapper for gdb. http://code.google.com/p/pygdb/.

[23] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

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

[25] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[26] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the USENIX Security Symposium*, 2001.

[27] smpCTF. smpctf 2010. http://ctf2010.smpctf.com/.

[28] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.