Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications

David Brumley, Pongsin Poosankam
{dbrumley,ppoosank}@cs.cmu.edu
Carnegie Mellon University

Dawn Song
dawnsong@cs.berkeley.edu
UC Berkeley & CMU

Jiang Zheng
jzheng@cs.pitt.edu
U. Pittsburgh

[*]

Abstract

The automatic patch-based exploit generation *problem is: given a program P and a patched version of the program P', automatically generate an exploit for the potentially unknown vulnerability present in P but fixed in P'. In this paper, we propose techniques for automatic patch-based exploit generation, and show that our techniques can automatically generate exploits for 5 Microsoft programs based upon patches provided via Windows Update. Although our techniques may not work in all cases, a fundamental tenet of security is to conservatively estimate the capabilities of attackers. Thus, our results indicate that automatic patch-based exploit generation should be considered practical. One important security implication of our results is that current patch distribution schemes which stagger patch distribution over long time periods, such as Windows Update, may allow attackers who receive the patch first to compromise the significant fraction of vulnerable hosts who have not yet received the patch.*

1 Introduction

At first glance, releasing a patch that addresses a vulnerability can only benefit security. We must, however, consider the entire time line for patch distribution. A new patch reveals some information, and having early access to a patch may confer advantages to an attacker. From a security standpoint, we should consider a) what information about a potentially unknown vulnerability is revealed by a patch, b) how quickly that information can be derived from the original and patched program, and c) what advantage that information yields to attackers. No previous work (such as fuzz testing as discussed in Section 7) has addressed these questions.

The *automatic patch-based exploit generation* (APEG) problem is: given a program P and a patched version of the program P', automatically generate an exploit for the potentially unknown vulnerability present in P but fixed in P'. Successful APEG would demonstrate that attackers could use patches to create exploits. To the best of our knowledge, APEG has not been previously demonstrated in public literature. Thus, the question of whether APEG is feasible for real-world programs was unanswered.

In this paper, we show that automatic patch-based exploit generation is possible as demonstrated by our experiments using 5 Windows programs that have recently been patched. We do not claim our techniques work in all cases or for all vulnerabilities. However, a fundamental tenet of security is to conservatively estimate the capabilities of attackers. Under this assumption, APEG should be considered practical, and those who have received a patch should be considered armed with an exploit.

One important consequence of our result is that having access to a patch confers a significant advantage over those who do not have access to the patch. The security advantage is important in light of current patch distribution practices. Current patch distribution practices stagger patch distribution, usually over hours, days, or longer. For example, Gkantsidis *et al.* show that for Windows Update it takes about 24 hours for 80% of the unique observed IPs to check for a new patch [18]. In

[*] This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422. Partial support was also provided by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and under grant DAAD19-02-1-0389 through CyLab at Carnegie Mellon. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, or the U.S. Government or any of its agencies. This work was also supported in part by the Korean Ministry of Information and Communication and the Korean Institute for Information Technology Advancement under program 2005-S-606-02.

All inputs
Safe Inputs

Figure 1. An input validation vulnerability occurs when the set of all inputs for P (in white) is a superset of the set of safe inputs for P (in black). The set difference is the set of exploits for P.

our experiments, we generate exploits from a patch in only a few minutes. Modern threats such as the Slammer worm have empirically demonstrated that once an exploit is available, most vulnerable hosts can be compromised in minutes [27]. Our results therefore imply that those who first receive a patch could potentially compromise most remaining vulnerable hosts before they receive a patch via current patch distribution architectures. Thus, our work indicates that current patch distribution schemes that stagger patch roll-out over large time periods requires rethinking.

Input Validation Vulnerabilities. We target *input validation vulnerabilities* where the set of inputs accepted by P is a superset of the safe inputs for P. Figure 1 shows this intuition graphically, where the set of safe inputs is a subset of all inputs for P. The difference between the safe and all inputs accepted by P is the set of exploit inputs. A common approach for patching such vulnerabilities is to add additional input sanitization checks in P' so that only safe inputs are processed without error. Many common types of vulnerabilities are at core input validation vulnerabilities, such as buffer overflows, integer overflows and underflows, and heap overflows.

Figure 2 shows a typical integer overflow input validation vulnerability we use throughout this paper. This example is motivated by a real-life vulnerability in Internet Explorer (called DSA_SetItem, for which we generate an exploit for in Section 4). All integers in this example are 32-bits, and therefore all arithmetic is performed mod 2^{32}. On line 1, the `input` integer variable is checked to see if it is even: if so, a temporary variable named `s` is assigned `input+2` (mod 2^{32}), else if odd, `input+3` (mod 2^{32}). Line 6 calls `realloc`, a manual memory management routine, which changes the size of the passed in `ptr` to point to `s` allocated bytes of memory. For example, if `s` is less than the size currently pointed to by `ptr`, then the resulting pointer will point to a smaller area of memory.

```
P: input is a user input
1. if(input%2 == 0) goto 2 else goto 4;
2. s := input+2;
3. goto 5;
4. s := input+3;
5. <nop>
6. ptr := realloc(ptr, s);
7. .. use of ptr ...;

P': input is a user input
1. if(input%2 == 0) goto 2 else goto 4;
2. s := input+2;
3. goto 5;
4. s := input+3;
5. if (s > input) goto 6 else goto ERROR;
6. ptr := realloc(ptr, s);
7. ... use of ptr ...
```

Figure 2. Our running example of an integer overflow input-validation vulnerability in P (top) and the patch P' (below). An integer overflow may happen on lines 2 or 4 of P. Line 5 of P' checks for overflow.

In this example, we consider any input that causes overflow on line 2 or 4 to be an exploit. Thus, the set of inputs which are exploits is $2^{32} - 3 \leq \texttt{input} \leq 2^{32} - 1$. At best, any exploit will cause a user of `ptr` after line 6 to cause a denial of service attack by crashing the program, or at worst, allow an attacker to hijack control of the program (as in the real-life vulnerability that motivated this example). The patched program P' adds a check for overflow on line 5. Any input which is an exploit for P will fail the inserted check in P'.

Challenges. One challenge for APEG is that software is often only available in binary (i.e., executable) form. Thus, in our approach and implementation, we target the case when P and P' are binary code. In our setting, P and P' can be either an executable programs or library. Addressing APEG for libraries is important since a) library vulnerabilities may often be exploited by multiple programs which use the library, and b) on many OSs security updates are often to libraries. For example, we conducted a survey of patches released from Microsoft in 2006 and found 84% of the security-related updates were changes in libraries. If P is a library, then the generated exploit x is a valid set of arguments to an exported (e.g., callable) function in the library, while if P is a program, x is an input to the program.

Another challenge is to isolate what changes have occurred between P and P'. To address this problem, security practitioners have developed tools, such as bindiff [33] and EBDS [13], which first disassemble both P and P', and then identify which assembly instructions have changed. Security practitioners use these differencing tools to help manually reverse engineer what the unknown or unpublished vulnerability that a patch addresses [13, 14, 31, 33], and in some cases, manually create exploits [14].

However, it is insufficient to simply locate the instructions which have changed between P and P'. In order for APEG to be feasible, one has to solve the harder problem of automatically constructing real inputs which exploit the vulnerability in the original unpatched program. Further, when feasible, it is important to know the speed at which exploits can be generated from patches in order to design adequate security defenses.

Approach Overview. Our approach to APEG is based on the observation that input-validation bugs are usually fixed by adding the missing sanitization checks. The added checks in P' identify a) where the vulnerability exists, and b) under what conditions an input may exploit the vulnerability. The intuition for our approach is that an input which fails the added check in P' is likely an exploit for P. Our goal is to 1) identify the checks added in P', and 2) automatically generate inputs which fail the added checks. In Figure 2, the goal would be to first discover the check added on line 5, then generate a value for `input` such that $P'(\texttt{input})$ that fails the check and leads to the `ERROR` state.

We call execution paths that fail the new check (i.e., execute the `ERROR` state in our example) in P' *exploitable paths* since any input that would execute such a path in P' is a likely exploit for P. There may be many exploitable paths, e.g., there are 2 exploitable code paths in our running example. However, the number of exploitable paths is typically only a fraction of all possible execution paths.

We propose techniques which scale when there are many different possible paths, but potentially only a few are exploitable. We present three different approaches: a dynamic analysis approach which considers a single path at a time, a static approach which encompasses multiple paths without enumerating them individually, and a combined approach based upon a combination of dynamic and static analysis. We show through evaluation that each technique is useful for automatically generating exploits from patches for different real-world vulnerabilities.

Results Overview. To evaluate the effectiveness of our approach, we have conducted experiments using 5 programs from Microsoft. Each program initially had a serious security vulnerability which was fixed by a patch. In some cases, the vulnerability is widely exploited, indicating the potential impact of future automatically generated exploits. Our results also show that each of the 3 approaches we propose have strengths for different vulnerabilities. In each case we are able to generate an exploit, usually within a few minutes. The fastest end-to-end time we were able to generate a verifiable exploit is under 30 seconds. We believe that with further work on our research prototype this time could be reduced.

In our evaluation, for the cases when a public proof-of-concept exploit is available, the exploits we generate are often different than those publicly described. We also demonstrate that we can automatically generate polymorphic exploit variants. Finally, we are able to automatically generate exploits for vulnerabilities which, to the best of our knowledge, have no previously published exploit.

Contributions. This paper shows that automatically generating exploits from patches within minutes should be considered practical. Current patch distribution architectures are not designed with the threat of APEG in mind. We argue that our results imply that we should immediately begin rethinking the design of current patch distribution architectures, and to this end, we propose several research directions.

Although we target the case where APEG is used by an attacker, APEG is also useful for security practitioners. For example, since APEG demonstrates a bug is exploitable, it could be used by vendors to prioritize bug fixes.

At the core of our approach for automatic patch-based exploit generation is the ability to generate an input that fails a check at a specified line of code. Generating inputs that execute a line of code is also studied in automatic test case generation. However, existing automatic test case generation techniques did not work for several vulnerabilities in our experiments. We propose a new technique based upon a mix of dynamic and static analysis to handle these cases. Thus, our techniques are likely to be of independent interest.

2 Automatic Patch-Based Exploit Generation: Problem Definition and Approach

2.1 Background Definitions

Our techniques are based on methods from the program verification community, thus we adopt their notation in this paper (such as in [11]). A program defines a relationship between an initial state space and a final state space. The state space of a program consists of all variables and memory. In our setting, memory is modeled as an array mapping 32-bit integers signifying memory addresses to 8-bit integers signifying memory values. In our setting, all registers are modeled as variables, and each memory cell can also be considered a separate variable when convenient. In Figure 2, the state space consists of memory and the variables `s` and `input`. When desired, we can also distinguish variables by their update site, e.g., the variable s on line 2 from s on line 5 (e.g., by transforming the program into static single assignment form [28]).

A *safety policy* ϕ is a first-order logic Boolean predicate from the programs state space to one of two values: safe or unsafe. In our setting, we consider only safety policies enforceable by an execution monitor [34]. At a high level, such policies are allowed to evaluate a boolean predicate on the program state space at each step of the execution, as well as keep track of any previous states executed so far. Common execution monitor enforceable safety policies include dynamic taint analysis, checking return address integrity, and dynamic type enforcement.

We denote executing P on input x as $P(x)$, and the execution of instruction i as $P_i(x)$. We denote checking the safety policy at execution step i as $\phi(P_i(x))$. The *vulnerability point* [5] for a vulnerable program is the first instruction i such that $\phi(P_i(x)) = \mathsf{unsafe}$.

We use the term *exploit* to mean an input x for which the safety policy returns unsafe. For example, if we use a dynamic taint analysis policy, an exploit would be any input that causes the analysis to raise a warning. One reason we use this definition of an exploit is that it does not presuppose a particular attack goal, e.g., information disclosure vs. denial-of-service vs. hijack control flow. This makes sense in our context since the vulnerability itself determines whether such specific attacks are even possible (e.g., information disclosure exploits are orthogonal to control hijack exploits). Note that there are potentially many different exploits, with each individual exploit called a *polymorphic variant*.

Safety policies are powerful enough (since they are first-order logic Boolean predicates over the entire program state space, including all memory) to specify specific kinds of attack when desired. For example, it is possible to specify a safety policy that is only violated by control hijack attacks. For example, we can create a safety policy which states the return address on the stack should not be overwritten by user input. Such a safety policy would only be violated by a typical control-hijack buffer overflow.

A program *control flow graph* (CFG) $G = (V, E)$ is a graph where each vertex $\in V$ is a single instruction, and there is an edge $(i_1, i_2) \in E$ if there is a possible transfer of control from instruction i_1 to i_2. An execution path is a sequence of vertices through the control flow graph such that for each vertex there is an edge to the next vertex in the CFG (note vertices may repeat in the path).

2.2 The Automatic Patch-Based Exploit Generation Problem

In the *automatic patch-based exploit generation problem*, we are given two versions of the same program P and P' where P' fixes an unknown vulnerability in P. The goal is to generate an exploit for P for the vulnerability fixed in P'. More formally, we are given a safety policy ϕ, and the programs P and P'. The purpose of ϕ is to encode what constitutes an exploit. Our goal is to generate an input x such that $\phi(P(x)) = \mathsf{unsafe}$, but $\phi(P'(x)) = \mathsf{safe}$.

2.3 Problem Scope and Approach

Vulnerabilities Addressed in this Paper. We focus on input validation vulnerabilities where user input is not sufficiently sanitized in P, but is sanitized via new checks in P'. Many common vulnerabilities are input validation vulnerabilities which are fixed by adding input sanitization logic. For example, if P is vulnerable to an integer overflow attack, then P' may insert a check for this overflow, and ultimately we will be using that inserted check to help derive an exploit. Another example is when P contains a typical buffer overflow where an input string may be too large, which is addressed in P' by inserting a check for overly-long inputs. However, a fix in which P' increases the size of the destination buffer to accommodate overly-long inputs currently falls outside our problem setting. We plan on targeting other types of vulnerabilities in future work.

Approach Overview. Our approach to APEG is based on the observation that the new sanitization checks added to P' often 1) identify the vulnerability point where the vulnerability occurs, and 2) indicate the conditions under which we can exploit P. Thus, an input x that fails the added sanitization check at the vulnerability point in P' is a *candidate exploit* for P. We call x a candidate exploit because a new check may not correspond to a real vulnerability. We verify a candidate exploit by checking $\phi(P(x))$, e.g., observing the execution of $P(x)$ within an execution monitor. Our approach therefore attempts to generate inputs which would fail the new checks inserted at the vulnerability point.

We can use off-the-shelf tools to identify the vulnerability point and the added checks. In our implementation, we use EBDS [13], a tool that automatically compares two executables and reports the differences. We can also use off-the-shelf safety checkers for ϕ. For example, dynamic taint analysis is a type of execution monitor commonly used to detect a wide variety of exploits.

Thus, in this paper, we focus on the technical challenge of how to automatically generate candidate exploits which reach and fail the given new checks in the patched version. To address this technical challenge, we propose an approach which 1) generates the set of constraints on the input domain to reach and fail the new check, and 2) finds a satisfying answer to the constraints, which is a sample candidate exploit.

More formally, we compute the *weakest precondition* [11] on the input state space of the P' to execute and fail the desired check. The weakest precondition is a constraint formula $\mathcal{F} : I \rightarrow \{true, false\}$ where I is the input state space, and a satisfying answer is our sample exploit. For example, the constraint formula

$$\mathcal{F}(\text{input}) \doteq \text{input}\%2 == 0 \land s = \text{input} + 2(\text{mod} 2^{32}) \\ \land \neg(s > \text{input})$$

is satisfied by all inputs that execute the true branch of P in Figure 2 and overflow. Finally, given the constraint formula, we query a solver to generate a satisfying answer to the formula (i.e., an input x such that $\mathcal{F}(x) = true$). If the solver returns a solution, the solution is a candidate exploit.

Thus, the steps to our approach are:

1. Identify the new sanitization checks added in P'. The remaining steps are performed for each new check individually (see Section 6 for a discussion on multiple checks).
2. Generate a candidate exploit x which fails the new check in P' by:
   (a) Calculating the weakest precondition to fail the new check in P'. The result is the constraint formula $\mathcal{F}$. We present three approaches for generating the constraint formula target this problem in Section 3.2.1.
   (b) Use a solver to find x such that $\mathcal{F}(x) = true$. x is the candidate exploit.
3. Verify a candidate exploit is a real exploit by running $\phi(P(x))$.
4. If desired, we can generate polymorphic variants. Let x be a known exploit. Let $\mathcal{F}'(X) = \mathcal{F}(X) \land (X <> x)$. Then x' such that $\mathcal{F}'(x') = true$ is a polymorphic variant exploit candidate. This process can be repeated to enumerate polymorphic variants.

3 Automatic Patch-Based Exploit Generation

In this section, we describe our approach and steps for automatic patch-based exploit generation.

3.1 Differencing Two Binaries Using an Off-The-Shelf Tool

The first step of our patch-based exploit generation is to difference P and P' to find new sanitization checks that are added in P'. Several tools exist for differencing binaries which are reasonably accurate and can be used to determine what new checks exist [12–14, 33]. We look for new checks that introduce a new code path since that indicates that P' is doing something different than P. We use eEyE's Binary Diffing Suite (EBDS) [13] in our implementation since it is freely available.

Our approach does not assume the differencer only outputs semantically meaningful differences (see Section 7). In fact, the differencer (EBDS) we use is based upon almost purely syntactic analysis of the disassembled binary. As a result, the list of new checks based on the syntactic analysis is a superset of the meaningful checks. Our approach will (correctly) fail to produce an exploit for semantically meaningless differences. For example, if P has the check $i > 10$, and P' has the check $i - 1 > 9$, the differencer may report the latter is a new check. Semantically meaningless differences such as these are weeded out by the verification step. For example, $i = 12$ is an example input which may satisfy the above difference, but would fail verification since it behaves the same in the new and old version. EBDS returns the list of differences; we filter them for new checks. EBDS also indicates whether the true or false branch of a new check corresponds to a new path. We assume a new path corresponds to failing the check. For example, in Figure 2 EBDS would report the false branch of the new check on line 5 introduces a new path, and we infer that $s >$ input is the check that should fail.

Recall that the remaining steps in our process of patch-based exploit generation are performed on each identified new check. Of course our approach benefits from better differencing tools which output fewer and more semantically meaningful checks, as fewer iterations are needed. In our evaluation, we measure the number of new checks reported by the tool, but assume the attacker can process each new check in parallel. This is realistic since attackers often have many (perhaps hundreds or thousands of) compromised hosts they can use for checking each reported difference.

If there is a need to prioritize which new checks are tried first for APEG, we have found that one effective scheme for prioritizing is to try new checks that appear in procedures that have changed very little. The eEye tool already provides a metric for how much a procedure has changed between P and P'.

3.2 Generating Constraint Formulas

In this section, we discuss techniques for automatically generating the constraint formulas. First, we explore the design space and provide intuition why we need to consider several different approaches. We then provide background on generating formulas using dynamic and static analysis (interested readers should consult the cited papers for full details). We then show how to adapt these ideas to the combined dynamic and static approach.

3.2.1 Key Design Points

The most important design question for constructing the constraint formula is to figure out what instructions to include in the formula. We need to include all the instructions for an exploitable path for the solver to generate a candidate exploit. However, the number of exploitable paths is usually only a fraction of all paths to the new check. Should the formula cover all such execution paths, some of them, or just one? We consider three approaches to answering this question: a dynamic approach which considers only a single path at a time, a static approach which considers multiple paths in the CFG without enumerating them, and a combined dynamic and static approach.

The Dynamic Approach: Generating a Constraint Formula from a Sample Execution. In some cases, the new check appears on a program path which is executed by a known input, e.g., along a commonly executed path. Such normal inputs can be found by examining logs of normal inputs, fuzzing, or other techniques. Of course, a normal input will likely satisfy the new check; otherwise, it is already a candidate exploit.

For such a given input i where $P'(i)$ executes the new check, we use techniques from dynamic analysis to generate the constraint formula representing the constraints on input for any execution of that single path up to the new check. Since the intuition behind our approach is that exploits fail the new check, we add an additional constraint that the input fails the new check.

The dynamic approach produces formulas that are typically the smallest of the three approaches. Since small formulas are generally the easiest to solve, the dynamic approach is usually the fastest for producing candidate exploits.

The ASPNet_Filter vulnerability in our evaluation (Section 4) is an example demonstrating real-world utility of the dynamic approach. In ASPNet_Filter, the vulnerability is in a webserver and the new check is added along a common code path which is executed by most URI requests. Thus, it is relatively easy to obtain at least one benign input that reaches the point of the new check, and hence it makes sense to start by analyzing that path first and see if we can generate an exploit using that path.

The Static Approach: Generating a Constraint Formula from a Control Flow Graph. Another approach is to create a formula over a CFG [6]. In particular, in the static case we are concerned with the CFG that includes all paths from the instruction where input is read to the new check. We perform program chopping on the program CFG in order to create a CFG that only includes paths to the new check. Computing a formula over the CFG is more efficient than computing a separate formula for each path in the CFG separately [6].

The static approach will generate a candidate exploit if any path in the CFG is exploitable. Since the static formula potentially includes all instructions in the CFG fragment, the formulas are typically larger and therefore take longer to solve. The DSA_SetItem vulnerability in our evaluation is an example where a purely static approach works.

Creating Constraint Formulas Using Combined Dynamic and Static Approach. If the CFG fragment contains a large number of instructions (because it covers a large number of paths), the generated formula may be too large for the solver. On the other hand, an exploit may never take the same execution path as a known input, thus a purely dynamic approach may not work either.

We propose a third approach which mixes the dynamic and static approaches to generating constraints. The intuition behind the combined approach is to combine information about code paths we know how to execute via known inputs, and additional code paths we wish to explore using static analysis. For example, we may know an input which does not reach the point of the new check, but does get us half-way there. We can use the dynamic analysis to the half-way point, then use the static approach for all paths from the half-way point to the new check.

The advantage of the combined approach is that it provides a way of considering a subset of paths so that the generated formula is (hopefully) small enough for the solver to generate a candidate exploit. The IGMP vulnerability in our evaluation is an example of this case where neither the static nor dynamic approach worked alone, but the combined approach generated a working exploit.

3.2.2 Background: Generating a Constraint Formula from a Sample Execution

Here we provide a recap of the overall method for generating a constraint formula from an execution trace. Due to space, interested readers should consult previous work [4, 5, 7, 19, 30] for a more thorough treatment.

The dynamic approach for creating a formula takes as input P', the new check, and a sample input i. We execute $P'(i)$ and record each instruction executed up to the sample check. We generate the constraint formula over the instructions executed along this path. To be efficient, we only record instructions (including all of their explicit and implicit operands) dependent upon in-

$$\frac{}{wp(x := e, Q) \vdash \text{let } x = e \text{ in } Q}\ \textsc{Assign}$$

$$\frac{}{wp(\text{assert } e, Q) \vdash e \wedge Q}\ \textsc{Assert}$$

$$\frac{wp(s_1, wp(s_2, Q)) \vdash Q_1}{wp(s_1; s_2, Q) \vdash Q_1}\ \textsc{Seq}$$

$$\frac{wp(s_1, Q) \vdash Q_1 \quad wp(s_2, Q) \vdash Q_2}{wp(\text{if } e \text{ then } s_1 \text{ else } s_2, Q) \vdash (e \Rightarrow Q_1) \wedge (\neg e \Rightarrow Q_2)}\ \textsc{Choice}$$

Table 1. Rules for calculating the weakest precondition.

puts since we only tackle vulnerabilities which can be exploited via user input.

Modeling the Executed x86 Instructions. In order to generate the constraint formula, we need to know the effects of each instruction executed. X86 is a complex instruction set. To accurately build the constraint formula, we need to model the effects of an x86 instruction correctly, including all implicit side effects such as updates to status registers. Thus, we raise the x86 instructions to an assembly modeling language we designed called Vine [2]. The ability to model the effects of each x86 instruction accurately is essential for automatically generating exploits.

We create a model of the trace by raising each instruction in the trace to Vine. We first lift each recorded instruction to Vine in a syntax-directed manner, e.g., if the x86 instruction `add eax, ebx` is in the trace, we produce the model statement `eax = eax + ebx`. Next, any operand which is not dependent upon input is replaced with its concrete value. Last, we assert that each branch condition in the trace will evaluate the same way as in the executed path.

Generating a Constraint Formula from the Modeled Path. The resulting execution trace from $P'(i)$ defines a single program path, which is also a valid model in Vine. The constraint formula is calculated over the straight-line model by calculating the weakest precondition [11]. We calculate the weakest precondition using the efficient algorithm and implementation given in Brumley *et al.* [6].

Table 1 shows the rules for calculating the weakest precondition. Each rule is read as an implication: if a program fragment matches the pattern shown below the horizontal bar to the left of the turnstile ($\vdash$), we perform the calculation shown on the top. The resulting formula is to the right of the turnstile. The rules inductively form an algorithm. The algorithm is initialized with $wp(P'(i), Q)$, where Q is a predicate that states the new check fails.

3.2.3 Background: Generating a Constraint Formula from a CFG

In the static approach, we raise all of P' to Vine as the first step. Since we are only concerned with paths that execute the new check, we remove Vine statements in the model for other paths. We achieve this by computing the chop [5, 6], and then constructing the constraint formula on the chop. Chopping is a technique which creates a smaller model that includes only those statements relevant to executing a sync node from a given start node in the CFG. In the static case, the start node is the input instruction, and the sync node is the new check. The exact algorithm we use for chopping is detailed in [5,6].

The formula we ultimately generate is over the CFG. Thus, a smaller, more compact CFG will generally lead to a smaller, easier-to-solve formula. In our experiments, the time to solve formulas usually dominates total exploit generation time, thus making formulas as easy as possible for the decision procedure to solve is important. Our experience has shown three common reasons formulas may take longer to solve: 1) "dead" code in the model where a value is computed but never used, 2) algebraic simplifications that can be performed, and 3) common sub-expressions that are recomputed. We have implemented common compiler optimizations on our modeling language to optimize the model: we remove dead code, perform as much algebraic simplification as possible, and remove redundant sub-expressions. In our evaluation, we show these optimizations can double the speed at which formulas are solved. (Note that these optimizations can also be applied in the dynamic case.)

The weakest precondition calculation used for the dynamic case applies equally well to any acyclic CFG [6]. We create an acyclic CFG by unrolling loops and recursive procedures a fixed number of times. Determining how many times to unroll a loop is known to be undecidable. In our evaluation, we unrolled loops only once.

The size of the generated formula is $O(n^2)$ in the number n of vine statements in the acyclic CFG [6]. Note that enumerating each path and applying a dynamic approach would result in a total formula $O(2^b)$ for b branches. Therefore, even though the static approach generates large formulas, it is more efficient than simply iterating the dynamic approach.

3.2.4 Formula Generation by Combined Static and Dynamic Analysis

Recall that the formula must cover all instructions for an exploitable path in order for the solver to generate a candidate exploit. The dynamic approach considers

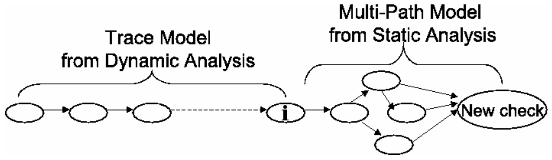

Figure 3. A graphical depiction of building a model of combined dynamic and static information.

only a single program path to the new check, but generates compact formulas and requires we know an input that executes the new check. The static approach covers more paths, but may produce larger formulas. At a high level, the only difference between the two is that the dynamic approach uses a trace to generate a straight-line program, over which we generate a formula, while the static approach uses the program to generate a branching acyclic program, over which we generate a formula. Thus, it should be of no surprise that the two can be *combined* where we alternatively combine the dynamic and static approach to select paths for formula generation. Although both the static and dynamic approach alone have been used previously to generate formulas, we are the first to propose the combined dynamic and static approach and demonstrate its feasibility in practice.

The high level intuition of a combined approach can graphically be represented as lolly-pop shaped, as shown in Figure 2. The combined approach offers a balance between the efficiency offered by single-path models produced by dynamic execution and the code coverage offered by multiple-path static techniques.

Suppose we have a trace containing executed instructions $0..n$. Let instruction $0 \leq i \leq n$ be a dynamic execution, and let there be a path from i to the new check, as shown in Figure 2. We build a combined model by first truncating the execution trace at instruction i to create the "stick" end. We create the lolly end by chopping off the program using the successor of i as the chop start and the new check as the chop sink. The two pieces are put together by adding the edge from i in the dynamic model to its successor in the static model. The resulting model considers only the straight-line program path up to i, then any subsequent path from i to the new check. We then compute the weakest precondition over the combined model.

The intuition why this works is that if we lifted the entire chop from instruction 0 to the new check, then the particular path taken by dynamic analysis is a path in the chop. Therefore, the path up to some step i in the dynamic trace to the chop is also a path. In the worst case, all paths from i to the new check are infeasible, i.e., there is no input that takes the path $0..i$ and then the successor $i + 1$ to the new check. Since the combined approach takes in two models and sequentially combines them, the result is a model.

For example, in our evaluation of the IGMP vulnerability, we combine an execution path that cannot be turned into an exploit with a chop of the procedure that contains the new check to create a combined model. Generating a formula and solving this model produces a working exploit for this example, but both the pure dynamic and static approaches do not.

Automatic Combined Execution. Automatic combined execution requires automatically deciding the mix point. In Figure 3, the question is which point should we choose as i. Of course one pre-requisite is we should choose an i such that there is a path in the static model from i to the new check. However, there still may be many such instructions in the trace.

One straight-forward approach is to take the i closest (in terms of CFG distance) to the new check and generate the combined model. If the formula generated on the combined model has no exploit, we pick instruction $i - 1$, and iterate.

In our experiments, we found a good heuristic that is quicker than the iterative approach is to choose i at procedure boundaries. Procedures are intended to perform a specific task independent of the remaining code. Therefore, by mixing at procedure points, the combined model includes overall tasks, instead of specific code paths. One implementation advantage of choosing procedure boundaries is that it is relatively straight-forward to implement automatic mixing: we simply set up a call to the static model of the procedure at the desired mix point in the trace.

3.3 Generating a Candidate Exploit from the Constraint Formula

We use STP [16], a decision procedure that supports bit-level operations, as a solver to generate candidate exploits from the constraint formula. When STP returns a satisfying solution for a given constraint formula, the solution provides a candidate exploit. By construction, the satisfying assignment will ensure that inputs taking on such satisfying assignment will make the program execution reach the point of the new check and fail the new check.

The need for bit-level support in the solver is necessary since assembly code typically makes use of bit-level operations such as $\oplus$ and logical shifts. For example, zeroing out a register r is usually not handled by a `mov` r, `0`, but by the equivalent `xor` r, r.

If the solver returns that there does not exist a satisfying solution for a given constraint formula, this means that it is not possible to have an input going down the paths covered in the constraint formula and failing the check. Thus, we need to build other constraint formulas covering other paths.

In some cases the solver may take too long to return an answer. In this case, we set a timeout and then move on to build other constraint formulas covering other paths. For example, the mix point can be changed so that fewer paths are included. In Section 4.4 we evaluate how the changing the mix point effects how long it takes the solver to generate a candidate exploit.

3.4 Generating Polymorphic Exploits.

Our approach allows us to enumerate (candidate) polymorphic exploit variants of the paths covered by $\mathcal{F}$. Suppose x satisfies $\mathcal{F}$. Let $\mathcal{F}'(X) = \mathcal{F}(X) \wedge (X <> x)$. $\mathcal{F}'$ is satisfied by all inputs except x that fail the check and execute a path in $\mathcal{F}$. Therefore a satisfying answer x' such that $\mathcal{F}'(x') = true$ is a polymorphic (candidate) exploit variant. This process can be repeated as desired.

3.5 Verifying a Candidate Exploit

We verify the candidate exploit x by checking if the safety policy ϕ is violated when executing $P(x)$. In our implementation, we use an off-the-shelf dynamic-taint-analysis-style exploit detector as a black box for ϕ for memory safety vulnerabilities. Using other types of exploit detectors is also possible. The candidate exploit is verified when the detector returns unsafe. If the verifier returns safe, and all paths to the new check have not been analyzed, then we iterate the above procedure on different code paths until an exploit is generated or all paths are exhausted.

3.6 Implementation

Our implementation of our three approaches for creating the constraint formulas is written in a mixture of C++ and OCaml. About 16,500 lines of C++ code is responsible for raising x86 to Vine. There are about 21,000 lines of OCaml. Most of the analysis, including chopping, code optimizations, and interfacing with the decision procedure is written in OCaml.

4 Evaluation

In this section, we evaluate our approach on 5 different vulnerable Microsoft programs which have patches available. Our experiments highlight that each approach for constraint formula generation — dynamic, combined, and static — is valuable in different settings. We show that we can generate exploits when no public exploit is available (to the best of our knowledge) for the ASPNet_Filter, IGMP, and PNG vulnerabilities. We also show that we can generate polymorphic exploit variants.

We focus on reporting our results on generating exploits for the new check which is exploitable, as discussed in Section 3.1. We also report the order in which the exploitable check would be found using the least-changed heuristic from Section 3.1.

4.1 Vulnerability and Exploit Description

DSA_SetItem Integer Overflow Vulnerability. The DSA_SetItem routine in comctl32.dll performs memory management similar to `realloc` [35]. The procedure takes in (essentially) a pointer p, a size for each object s, and a total number of objects n. The procedure calls `realloc(p, s*n)`. An overflow can occur in the multiplication $s*n$, resulting in a smaller-than-expected returned pointer size. Subsequent use of the pointer at best causes the application to crash, and at worst, can be exploited to hijack control of the application. `DSA_SetItem` can be called directly, or indirectly by a malicious webpage via the `setSlice` JScript method. In practice, this vulnerability is widely exploited on the web either by overtly malicious sites and legitimate but hacked web sites [29].

The patched version adds logic to protect against integer overflow. In particular, it adds a check that overflow never happens and the result is $< 2^{31}$ (i.e., always positive).

EBDS took 371.9 seconds to perform the diff. 21 functions were found changed, and 5 new functions were added. Given the least-changed heuristic, the exploitable check would be the 3rd check tried.

Exploit Generated: The exploits we generated caused a denial of service attack, e.g., Internet Explorer crashed. Any ϕ that can detect pointer misuse is suitable: we used TEMU [2]. We also could specify specific memory locations to overwrite. Determining the specific address for a successful control hijack requires predicting the processes memory layout, which changes each time the process is invoked. Attackers currently do this by essentially repeatedly launching an attack until the memory layout matches what the exploit expects. We similarly repeatedly launch the attack until we achieve a successful control hijack.

ASPNet_Filter Information Disclosure Vulnerability (MS06-033; Bugtraq ID#18920; CVE-2006-1300). The ASPNet_Filter DLL is responsible for filtering ASP requests for the Microsoft .NET IIS Server, and is vulnerable to an information disclosure attack. The module filters sensitive folder names from a URI request during processing so that information contained in these folders is not disclosed upon response. These folders are automatically built using ASP.NET's default template. For example, `App_Data`, `App_Code`, and `Bin` are used to store data files, dynamically compiled code, and compiled assemblies, respectively. An exploit for this vulnerability would allow the attacker to view files under these folders. This is a serious vulnerability because scripts in these directories often contain sensitive information, such as passwords, database schemas, etc. To the best of our knowledge, there are no public exploits for this vulnerability.

The unpatched version performs proper filtering for URI requests that use forward slashes ('/'), but not backslashes ('\'). The patched version fixes this vulnerability by checking for '\' and flipping them to '/'.

EBDS took 16.6 seconds to perform the diff. One new function was added, along with 4 changes to existing procedures to call the new function. The exploitable check using the least-changed heuristic would be the first one tried.

Exploit Generated: The exploit we generated was able to read files in the protected directories. Currently we do not have implemented a ϕ that detects such attacks, so we verified the generated candidate exploit manually.

IGMP Denial of Service Vulnerability (MS06-007; Bugtraq ID#16645; CVE-2006-0021). The IGMP (Internet Group Management Protocol) protocol is used for managing the membership of multi-cast groups. An exploit for this vulnerability is an IGMP query packet with invalid IP options. The invalid options can cause the IGMP processing logic to enter an infinite loop. Since IGMP is a system-level network service, an exploit will freeze the entire vulnerable system. The patch adds checks in the IGMP processing routine for invalid IP options. To the best of our knowledge, there is no public exploit for this vulnerability. [1]

EBDS took 157.08 seconds to diff the patched and unpatched tcpip.sys. The diff identified that one function was changed. Using the least-changed heuristic, the exploitable check would be first.

The exploit we generated successfully caused the denial-of-service. Currently we do not have implemented a ϕ that detects deadlock due to an infinite loop, thus we verified our candidate exploit manually.

GDI Integer Overflow Vulnerability (MS07-046; Bugtraq ID#25302; CVE-2007-3034). The Windows Graphic Device Interface (GDI) is the core engine for displaying graphics on screen. The GDI routine responsible for showing metafile graphics is vulnerable to an integer overflow. The integer overflow can subsequently lead to a heap overflow, which at best causes a system crash, and at worst, can result in a successful control hijack.

The patch addresses the integer overflow by adding 5 additional checks when loading a metafile. The unpatched version is exploitable when any one of the 5 checks fails.

EBDS took 109 seconds to diff the patch and unpatched version. The diff identified the 5 additional checks. Since an exploit can fail any of the 5 checks, an exploitable check would be tried immediately using the least changed heuristic.

Exploit Generated: The exploit we initially generated caused a denial-of-service. This vulnerability is similar to DSA_SetItem: we can specify what to overwrite in the heap structure, but the location of the heap structure depends upon the process layout. Thus, a successful control hijack required repeatedly launching the attack. Any ϕ that detects pointer misuse is appropriate: we used TEMU [2].

PNG Buffer Overflow Vulnerability (MS05-025; Bugtraq ID#13941; CAN-2005-1211). PNG (Portable Network Graphics) is a file format for images utilized by many programs such as Internet Explorer and Microsoft Office programs. Each PNG image contains a series of records which specify different properties of the image, e.g., whether the image is indexed-color or gray-scale, the alpha channel, etc. In the indexed-color mode, the record format specifies an additional alpha channel byte value for each indexed color. A heap-based buffer overflow occurs in early Microsoft implementations when the number of alpha channel bytes exceeds the number of pre-specified colors.

The patched version adds additional checks to validate PNG record fields. To the best of our knowledge, there are no public exploits for this vulnerability.

The total time to diff the two vulnerable versions was 27.05 seconds. Changes were only reported in the vulnerable procedure, with the exploitable check being the first using the least changed heuristic.

Exploit Generated: The exploit we generated initially caused the program to crash, similar to GDI and DSA_SetItem. Again, we use TEMU [2] to confirm candidate exploits, but any ϕ that detects pointer misuse is also possible. This attack is on the heap, and also required us to repeatedly launch the attack to achieve successful control hijack.

4.2 Patch-Based Exploit Generation using Dynamic Analysis

We successfully generated exploits for the DSA_SetItem, ASPNet_Filter, and GDI vulnerabilities using dynamic analysis. For DSA_SetItem, we recorded the execution trace of IE 6 loading a valid

[1] An EBDS [13] tutorial discusses this vulnerability. However, they do not create an exploit.

	DSA_SetItem	ASPNet_Filter	GDI
Trace	4.99	4.50	9.92
Formula	0.52	0.14	0.41
Solver	0.17	6.93	0.01
Total	5.68	11.57	10.34

Table 2. Time to generate an exploit using the dynamic approach. All times are in seconds.

	DSA_SetItem		GDI	
	no opt	opt	no opt	opt
Model Gen	1.35	1.45	3.61	3.97
Formula	2.48	0.87	3.45	1.02
Solver	182.91	81.15	19.61	21.42
Total	186.74	83.47	26.67	26.41

Table 3. Time to generate exploit using the static approach. All times are in seconds.

webpage that calls the setSlice ActiveX control method, which in turn calls DSA_SetItem. For ASPNet_Filter, we recorded IIS processing an HTTP request from a log file. For GDI, we created an image within a PowerPoint presentation, then saved the image in the Windows metafile format. We recorded the execution of a small GDI application loading the saved file. All execution traces were recorded using TEMU [2].

Table 2 shows an overview of our results. All times in the table are in seconds. The "Trace" row shows the amount of time it took to generate a trace using TEMU. The "Formula" row shows the amount of time to lift the trace to our modeling language and produce the constraint formula. The "Solver" row indicates how long it took the solver to solve the formula.

The total time to generate an exploit after diffing is under 12 seconds in all experiments. If we include diffing time, then the total exploit generation time for DSA_SetItem is 377.58 seconds, ASPNet_Filter is 28.17 seconds, and GDI is 119.34 seconds.

We were not able to generate exploits using the dynamic approach for the IGMP and PNG vulnerabilities. For IGMP, we recorded the execution of Windows processing the sample IGMP message from [10]. The identified new checks were executed. However, the constraint formula built was not satisfiable by any input that failed the new check. The reason is that the particular execution path taken was already constrained so the added check could never fail (i.e., was redundant along that path). For PNG, we were not able to generate an exploit for a sample execution trace for the same reason: the path constraints prevented the new check from ever failing. In particular, the execution of PNG involves the calculation of a CRC-32 checksum. There were no other inputs along the chosen path that satisfied the checksum while failing the new check.

4.3 Patch-Based Exploit Generation using Static Analysis

We were able to generate exploits for the DSA_SetItem and GDI vulnerabilities using a purely static approach. For DSA_SetItem, the static model included setSlice and DSA_SetItem. For GDI, the vulnerable procedure `GetEvent` is reachable by the explored API `CopyMetaFileW`. Thus, our static model consisted of these two functions.

Table 3 shows an overview of our results. All times in the table are in seconds. We include in this table the time to generate a model of all static paths to the new check under the "Model" row. For each vulnerability, we also consider two cases: with and without the optimization on the model discussed in Section 3.2.3.

Without optimization, we were able to generate exploits for DSA_SetItem in 186.74 seconds. When we enable optimizations, the time to generate the model increases, but the subsequent steps are much faster. In particular, the optimizations for DSA_SetItem reduce the time to generate an exploit from the formula by about 55%. We believe further optimizations would likely further reduce the solution time. For GDI, the optimizations had little effect, saving only .26 seconds overall.

We enumerated 3 different exploits for the DSA_SetItem vulnerability. In particular, we enumerated both the public exploit, and 2 new exploit variants.

One way to compare the advantage of the static approach is to measure the number of paths to the new check included in the formula. A similar formula using the dynamic approach alone would require enumerating each path. There are 6 exploitable paths to the new check for DSA_SetItem in the static model we consider. There are about 1408 total paths in the static model for the GDI vulnerability.

We were not able to generate exploits statically for the PNG, IGMP, and ASPNet_Filter vulnerabilities. In the ASPNet_Filter vulnerability, there are system calls

	DSA_SetItem	IGMP	GDI	PNG
Trace Gen	4.99	10.14	9.92	103.28
Model Gen	1.42	2.58	3.36	0.58
Formula	0.31	12.57	.027	0.28
Solver	4.79	3.78	0.26	0.14
Total	11.51	29.07	13.57	104.28

Table 4. Time to generate an exploit using the combined approach. All times are in seconds.

Dyn:Static	Formula Size	Solver Time	# Paths
4:1	309250	18.94	496
3:2	310414	22.77	496
2:3	6549513	Out of Mem	10416

Table 5. Results for changing the mix point at different points in the call path to the vulnerable procedure. The formula size is the number of expressions in the formula. Solver time is in seconds.

not currently supported by our constraint formula generator. The standard solution is to generate summaries of the effects [6, 8]. A manual analysis indicates that simply omitting the various calls would likely still result in a formula that generates exploits. We leave exploring such extensions as future work. We could not generate exploits for all paths statically for the PNG and IGMP vulnerabilities because the solver ran out of memory trying to solve the generated constraints.

4.4 Patch-Based Exploit Generation using Combined Analysis

We successfully generated exploits using the combined approach for DSA_SetItem, IGMP, GDI, and PNG. In our experiments, we use the heuristic to mix at procedure boundaries.

Table 4 show our results when we mix using the dynamic trace from Section 4.2 up to the vulnerable procedure. The static approach generates a formula for the vulnerable procedure. The two are then spliced together.

The mixed approach works for IGMP and PNG, but the purely dynamic and purely static approaches do not. In both cases the purely dynamic approach fails because the executed path in the trace is not exploitable. In both cases the static approach also fails because the solver runs out of memory. The combined approach offers a way to build a formula for a subset of potentially exploitable paths without enumerating them individually.

We also measured how mixing reduces the static formula size for the IGMP vulnerability. The shortest call path to the vulnerable function has length 5: IPRcvPacket → DeliverToUserEx → DeliverToUser → IGMPRcv → IGMPRcvQuery. We consider mixing at IGMPRcvQuery, IGMPRcv, and DeliverToUser, i.e., the formula consists of all paths through 1, 2, and 3 procedures, and the rest from the dynamic path.

Table 5 shows our results. This table shows that using the dynamic formula for IPRcvPacket → DeliverToUserEx → DeliverToUser and the static for IGMPRcv and IGMPRcvQuery is solvable, while adding all paths for DeliverToUser creates a formula that is too difficult to solve. It also shows a common behavior when solving formulas in our experience: they are either solvable relatively fast, e.g., within a few minutes, or they are not solvable within a reasonable amount of time.

5 Implications of Automatic Patch-Based Exploit Generation

Our evaluation demonstrates APEG for several vulnerabilities. Since we must conservatively estimate the capabilities of attackers, we conclude APEG should be considered a realistic attack model. The feasibility of automatic exploit generation has important implications on the security landscape. One of the most immediate problems is rethinking today's patch distribution practices in light of these results.

In today's patch distribution practices, vulnerable systems typically download patches at different times, creating a time window from when the first vulnerable system downloads a patch to the last. Staggered patch distribution is attractive because it prevents huge traffic spikes when a new patch is released. For example, recently Gkantsidis *et al.* conducted a large scale study of users of Microsoft Update. Their measurements show that it takes about 24 hours for Windows Update to see 80% of the unique IPs of hosts checking for a patch [18].These measurements confirm the intuition that not everyone will receive a patch at the same time, with gaps of hours if not longer before even the majority receive the update.

In our results, we are typically able to create exploits from the patch in a matter of minutes, and sometimes seconds. Therefore, APEG could enable those who first received a patch to generate an exploit and compromise a significant fraction of systems *before they even had a chance to download the update.* Note this is irrespective of whether people actually apply the patch; but whether they even have the opportunity to apply it.

There are many approaches to fix staggered patch distribution. We discuss three directions: 1) make it hard to find new checks (through obfuscation), 2) make it so everyone can download the update before anyone can apply it (using encryption), and 3) make it so everyone can download the patch at the same time (using P2P).

Patch Obfuscation. One approach is to hide what lines of code changed between P and P'. In particular, vendors could obfuscate patches such that the difference between P and P' is very large. This approach would be the easiest to break our particular implementation, since the results of EBDS [13] would contain too many instructions to isolate which checks were added.

The advantage of this approach is obfuscation techniques are widely available. However, there are many challenges to the obfuscation approach. For example, figuring out the level of obfuscation necessary to thwart attackers may be tricky. Simple instruction replacement, e.g., multiplications by 2 with left shifts, may thwart EBDS but not a more sophisticated tool that focused on semantic, not assembly-level syntactic differences. Another problem is the effects of obfuscation should be transparent to legitimate users, e.g., obfuscation that degrades performance is likely unacceptable.

Patch Encryption. We could initially encrypt patches so that simply having the patch leaks no information. Then, after a suitable time period, a short decryption key (e.g., 128-bits) is broadcast. This scheme allows all users who have the patch and receive the key to apply it simultaneously. Others have independently arrived at similar ideas [32].

Patch encryption allows vendors to use essentially the same staggered patch distribution architecture while defending against automatic patch-based exploit generation. Simultaneously (or near simultaneously) distributing the decryption key is possible since the key is very small, e.g., 64-bits. Therefore, this scheme is potentially fair in the security sense: everyone has the same opportunity to apply the patch before anyone could potentially derive an exploit. However, one potential problem is how to handle off-line hosts. A second problem is the actual fixes are delayed from the users perspective, which raises a number of policy issues. There are security-related policy choices, e.g., should patches be encrypted when a zero-day exploit is available to a few attackers, but not all attackers. There are also human-related choices, e.g., people may not like the idea of having a patch that they cannot apply. Further research is needed to answer such questions.

Fast Patch Distribution. It may be possible to change patch distribution so everyone receives the patch at about the same time. For example, Gkantsidis *et al.* propose using a peer-to-peer network for patch distribution in order to reduce the load on patch distribution servers [18]. Such a peer-to-peer system could potentially also distribute patches faster than the centralized model. However, such as scheme would still need to address off-line hosts. It is also unclear whether such a scheme is fast enough to combat APEG.

6 Discussion

Generating Specific Exploits. The techniques we describe generate an exploit from the universe of all exploits for a patched vulnerability. At a high level, the solver gets to pick any exploit that satisfies the generated formula. We can make the formula specific to achieve a particular attack purpose, e.g., a control hijack attack.

Note that since initially we do not know what vulnerability is patched, it does not make sense to try to create a specific type of exploit *a priori*. For example, if the unknown vulnerability is an information disclosure vulnerability, it makes no sense to try to create a control hijack exploit.

However, once we know what vulnerability can be exploited, we can extend our approach to generate specific kinds of attacks, as long as we can write the conditions necessary in the modeling language. Vine allows us to specify meta-properties we would like to hold on the x86 program. Thus, we can state a meta-property such as asserting a store instruction overwrites the return address. For example, the x86 call instruction is modeled as first storing the return address on the stack, then jumping to the designated program location. An x86 return instruction can be modeled as loading a 32-bit number from the stack, then jumping to the given address in Vine. In the modeling language, we can add checks about the x86 program such as the return is to the same address stored by the call instruction. Overwriting the stack pointer is just one example: we could monitor the initial exploit to garner more information about what sensitive data structures are possible to overwrite. We leave exploring this as future work.

Dealing with Multiple Checks. The patch for a single vulnerability may have many new checks in the patched version. In some cases, our techniques will still work as in the GDI vulnerability. In other cases, it is not so clear. Recall that the model is generated with respect to the patched program. Consider the case where an input has to fail two new checks a and b in sequence to exploit the unpatched version. Initial exploit generation for a may generate an exploit, but verification will fail since by assumption the program is not exploitable when check a fails alone. We then consider b. Since we are building a model over the patched program, the model represents all potential paths through a and b, e.g., the case where they fail together, but also the case where a succeeds but b fails. By default the formula generated by our techniques considers each check independently. Since the set of inputs which fail b and a is a subset of those that just fail b, we may get lucky and the decision procedure returns an input which fails both a and b. From the security standpoint, it is usually prudent to assume attackers are lucky. However, we may also get back an answer where b fails but a does not, since that is all the formula required. This can be solved by querying for various combinations of new checks. Since considering each combination is undesirable, this problem would benefit from further research.

Note an independent problem is if an update addresses multiple vulnerabilities. Since our current approach is considers each check individually, it would simply be iterated over all checks irrespective of how many vulnerabilities are patched.

Other Applications of Our Techniques. Our techniques have applications in other areas. For example, automatic deviation detection is concerned with the problem of finding any input i for programs P_1 and P_2 such that the behavior of $P_1(i)$ is different than $P_2(i)$. In our scenario, $P_1 = P$ and $P_2 = P'$, and the deviation input $i = e$ such that P_1 is exploited but P_2 is not. Previous work focused on deviation detection from a single dynamic trace [4]; we consider multiple paths.

We expect our techniques, especially combined dynamic and static formula generation, will be applicable to many similar problems that require modeling multiple program paths. Most previous work that requires generating a formula to represent a program path only focus on a single path for scalability reasons. Our work shows for the first time that scaling up to multiple paths is possible. In particular, applying the combined static and dynamic approach to other settings is an interesting avenue to explore.

7 Related Work

Fuzzing to find inputs which crash programs essentially tries random or guided semi-random inputs on a program [15, 20, 24–26]. Fuzzing tools have recently become popular as a way of finding exploits for programs, e.g., fuzzing found numerous vulnerabilities in the Month of Browser Bugs [1]. Recently, fuzzing techniques have been augmented to produce particular kinds of exploits, e.g., control-hijack exploits for buffer overflow vulnerabilities [24]. Unlike fuzzing, our approach is goal-oriented: we find an input that reach a specific line of code (the new check). Instead of searching for vulnerabilities at random, we use the patch as a guide to generate exploits. Fuzzing and similar techniques also only consider P, thus do not address generating exploits from patches.

We use an off-the-differencer to identify changes. Research in finding semantic differences, such as BinHunt [17], would help winnow down the number of new checks for which we try exploit generation.

Our techniques are closely related to automatic test case generation, which has a long history (e.g., [3, 21–23]). Our techniques are most closely related to goal-based test generation (e.g., [21]) where inputs are automatically generated that will execute a given goal statement in the program. Test case generation does not address the problem of creating exploits from patches, and therefore does not address the security ramifications.

Similar techniques for generating formulas in the static and dynamic approaches have previously been applied to signature generation [5,6,8,9]. We use the chopping algorithm from our previous work [5], and generate formulas using the efficient method from [6].

8 Conclusion

We have demonstrated that automatic patch-based exploit generation is possible in several real-world cases. In our evaluation, we are able to automatically generate an exploit given just the unpatched and patched program usually within a few minutes. In order to achieve our results, we developed novel techniques for analyzing potential exploitable paths to a new sanitization check. Since best security practices dictate that we conservatively estimate the power of an attacker, our results imply that in security critical scenarios automatic patch-based exploit generation should be considered practical. One immediate consequence we suggest is that the current patch distribution schemes are insecure, and should be redesigned to more fully defend against automatic patch-based exploit generation.

Acknowledgements

The authors would like to thank the anonymous referees, Ivan Jager, James Newsome, Steven Rudich, Vyas Sekar, and Shobha Venkataraman for their feedback in preparing this paper.

References

[1] Month of browser bugs website. `http://browserfun.blogspot.com`, 2006.

[2] The BitBlaze binary analysis project. `http://bitblaze.cs.berkeley.edu`, 2007.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ACM Internation Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Symposium*, Boston, MA, Aug. 2007.

[5] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[6] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2007.

[7] 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.

[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] A. Crosswell. Igmp v3 tcpdump trace. `http://www.columbia.edu/~alan/igmp/ex1b/`.

[11] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[12] T. Dullein and R. Rolles. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de L'information et des communications*, 2005.

[13] eEyE Security. eEye binary diffing suite (EBDS). `http://research.eeye.com/html/tools/RT20060801-1.html`. Version 1.0.5.

[14] H. Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2004.

[15] J. Forrester and B. Miller. An empirical study of the robustness of windows nt applications using random testing. In *4th USENIX Windows Systems Symposium*, 2000.

[16] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Proceedings on the Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, Berlin, Germany, July 2007. Springer-Verlag.

[17] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. Technical report, School of Information Sciences, Singapore Management University, February 2008.

[18] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnovic. Planet scale software updates. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2006.

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

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

[21] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2):1998, 1998.

[22] N. Gupta, A. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. *ACM SIGSOFT Software Engineering Notes*, 23(6):231–244, Nov. 1998.

[23] B. Korel. Automated test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990. path-based test set generation.

[24] J. Medeiros. Automated expoit development: The future of exploitation is here. `http://toorcon.org/2007/talks/19/toorcon_whitepaper.pdf`, 2007.

[25] B. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the International Workshop on Random Testing*, 2006.

[26] 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.

[27] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. In *Proceedings of the IEEE Symposium on Security and Privacy*, volume 1, 2003.

[28] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[29] R. Naraine. Crime rings target ie 'setslice' flaw. `http://www.eweek.com/article2/0%2C1759%2C2022805%2C00.asp`, 2006.

[30] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In R. Write, S. D. C. di Vimercati, and V. Shmatikov, editors, *Proceedings of the ACM Conference on Computer and Communications Security*, pages 311–321, 2006.

[31] A. Protas and S. Manzuik. Skeletons in microsoft's closet. BlackHat Europe 2006: `http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Manzuik.pdf`.

[32] J. Roskind. Attacks against the netscape browser plus security response philosophy and methods. Private communication and seminar talk.

[33] Sabre Security. Bindiff. `http://www.sabre-security.com/products/bindiff.html`.

[34] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[35] Secure Science Corporation. Analysis of the WebViewFolderIcon ActiveX integer overflow (setSlice). `http://www.mnin.org`, 2006.