Software Crash Analysis for Automatic Exploit Generation on Binary Programs

Shih-Kun Huang, *Member, IEEE*, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai

***Abstract*—This paper presents a new method, capable of automatically generating attacks on binary programs from software crashes. We analyze software crashes with a symbolic failure model by performing concolic executions following the failure directed paths, using a whole system environment model and concrete address mapped symbolic memory in $\mathbf{S^2E}$. We propose a new selective symbolic input method and lazy evaluation on pseudo symbolic variables to handle symbolic pointers and speed up the process. This is an end-to-end approach able to create exploits from crash inputs or existing exploits for various applications, including most of the existing benchmark programs, and several large scale applications, such as a word processor (*Microsoft office word*), a media player (*mpalyer*), an archiver (*unrar*), or a pdf reader (*foxit*). We can deal with vulnerability types including stack and heap overflows, format string, and the use of uninitialized variables. Notably, these applications have become software fuzz testing targets, but still require a manual process with security knowledge to produce mitigation-hardened exploits. Using this method to generate exploits is an automated process for software failures without source code. The proposed method is simpler, more general, faster, and can be scaled to larger programs than existing systems. We produce the exploits within one minute for most of the benchmark programs, including *mplayer*. We also transform existing exploits of *Microsoft office word* into new exploits within four minutes. The best speedup is 7,211 times faster than the initial attempt. For heap overflow vulnerability, we can automatically exploit the *unlink()* macro of *glibc*, which formerly requires sophisticated hacking efforts.**

***Index Terms*—Automatic exploit generation, bug forensics, software crash analysis, symbolic execution, taint analysis.**

Acronyms and Abbreviations

AEG	Automatic Exploit Generation
API	Application Programming Interface
ASLR	Address Space Layout Randomization
EAX	Extended Accumulator Register
EBP	Extended Base Register
Concolic	Concrete and Symbolic Execution
CVE	Common Vulnerabilities and Exposures
DoS	Denial of Service
EIP	Extended Instruction Pointer
ELF	Executable and Linking Format
GOT	Global Offset Table
IDS	Intrusion Detection System
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
LOC	Lines of Code
NOP	No Operation
ROP	Return-Oriented Programming
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language
TCG	Tiny Code Generator
OS	Operating System
PC	Path Conditions
POSIX	Portable Operating System Interface
XSS	Cross Site Scripting

Notations

d_i	The symbolic read data object.
$d_i.address$	The symbolic address expression.
$d_i.value$	The pseudo symbolic variable.
$d_i.memory$	The memory snapshot.
D	All Dereference objects during the execution of the state.
$bExpr$	The Boolean expression.
A	The mapping that maps each element d_i of D to a concrete address $A(d_i)$.
$V(e)$	The value of e under the variable assignment V.

Manuscript received September 02, 2012; revised April 18, 2013; accepted May 20, 2013. Date of publication January 20, 2014; date of current version February 27, 2014. This work was supported in part by NCP, TWISC, the National Science Council(NSC-101-2221-E-009-037-MY2, and NSC 100-2219-E-009-005), and the Industrial Technology Research Institute of Taiwan (ITRI FY101 B3522Q1100).

S.-K. Huang is with the Information Technology Service Center, National Chiao Tung University, Hsinchu 30010, Taiwan (e-mail: skhuang@cs.nctu.edu.tw). Associate Editor: S. Shieh.

M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai are with the Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan (e-mail: mhhuang@cs.nctu.edu.tw; luhl@cs.nctu.edu.tw; huangpy@cs.nctu.edu.tw; laicw@cs.nctu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2014.2299198

I. Introduction

CRAFTING exploits for control flow hijacking, SQL injection, and cross-site scripting (XSS) attacks is typically a manual process requiring security knowledge [1]. However,

based on recent advances in symbolic execution, several prototype approaches to automatically generating exploits have been proposed [2]–[4]. Exploits or other types of attacks, e.g., SQL injection, and XSS [5], have been used for auditing web application security, IDS signature generation, and attack prevention. These research topics belong to dynamic taint analysis and symbolic execution.

The motivation of this work is straightforward. Failure of software including web applications is inevitable. Given a large number of failures, we need a systematic approach to judge whether they are exploitable. In Miller *et al.*'s crash report analysis [6], the authors analyze crashes by BitBlaze [7]. Compared with !exploitable [8], the results show that exploitable crashes could be diagnosed in a more accurate way although still with limitations and requiring manual efforts. Moreover, crash analysis plays an important role to prioritize the bug fixing process [9]. A proven exploitable crash is surely the top priority bug to fix.

Dynamic taint analysis and forward symbolic execution have been the primary techniques in security fields [10]. Q [4] is a recent success to generate mitigation-hardened ($W \oplus X$ and ASLR [11]) exploits by feeding concrete execution trace and triggering a tainted instruction pointer. The exploits divert the vulnerable path by concolic execution, and exploit constraints to manipulate the instruction pointer. The constraints combine with return-oriented programming (ROP) [12] payload, and feed to a decision procedure, STP [13].

Our objective is similar to Q, but we target a new threat model. Note that most threats are continuations [14], [15]. Continuations are the explicit control abstraction to express "what to do next" or "what remains of a computation" to give a formal modeling of the goto instruction. They can be used to program non-local jumps, exception handling, and user threads. We can thus view software security threats as attackers' explicit control on the victim programs. Attackers give explicit control of "what to do next" from the original running software, and take control of the rest of a program's computations. For example, control flow hijacking attacks divert the input into an attacker manipulated continuation. The continuation results in the execution of arbitrary code. The SQL and command injections are inputs flowing into the SQL server or introducing a shell command execution. They are continuations into the SQL or command shell context. The cross-site scripting is a reflection of web pages, inserting an explicit continuation to execute arbitrary Javascripts, impersonating as originating from the original Web server. If the continuation is symbolic, which is a concolic execution to reach the invoked site of the continuation and a symbolic expression to describe the continuation, we can generate practical attacks to exploit the continuation. A software crash can be viewed as a tainted continuation. Furthermore, if the tainted continuation is symbolic, an exploit can be automatically generated. We have successfully produced exploits from software crashes for control flow hijacking attacks from large applications, including *Mplayer*, *Unrar*, *Foxit* pdf reader, CMU's AEG [2], and MAYHEM [16] benchmarks. We have also produced exploits (with our own shellcode) from existing exploits of Microsoft Office Word. All processes are end-to-end, built on top of the environment model of $\mathrm{S^2E}$ [17] (based on KLEE [18], and QEMU [19]).

The framework, called CRAX, is to act as a backend of static and dynamic program analyzers, bug finders, fuzzers, and a crash report database. Given these software failures from the frontends and the program binary, CRAX can automatically generate attacks, and practical mitigation-hardened exploits.

The primary contributions and impacts of the work are as follows.

- Address exploit generation for large software systems without source code. Concolic execution ideas have been the techniques for exploit generation since 2009. Automatic exploit generation has become an integration effort from existing systems, due to the rapid development of symbolic computation, processor emulation, and environment model supports. However, we have not found practical integration of exploit generation work that can produce exploits from large applications, such as MS-office word, Mplayer, and Foxit pdf reader. We are the first to demonstrate such a capability though completing the exploiting process formerly regarded as a manual process. A similar scale of work is that of Catchconv [20], which performs symbolic fuzzing, taking the Mplayer as a prey. However, it only acts as a fuzzer, and succeeds to produce Mplayer crashes about as frequently as was done in zzuf [21]. In contrast, CRAX takes the crash from Mplayer and produces exploits. We have automated the exploit writing process of large binary programs.
- Prioritize crashes to be fixed. Currently, several sources of crash reports from several bug analyzers and random fuzzers are available. Too many crashes need to be fixed, and there is a pressing need to determine their priorities [9]. We have preliminarily examined the crash database for open source projects, and found that many of the crashes can be the seed input to produce exploits. The tool would be the first screen gateway to prioritize the bug fixing order.

We organize the paper as follows. Sections II and III describe the exploit generation method and basic implementation. We discuss the concept of pseudo symbolic variables for dealing with symbolic pointers in Section IV, and the format string exploit generation process in Section V. Section VI reports the experimental results. Section VII presents related work. We conclude this paper in Section VIII.

II. Exploit Generation Method

We model the exploit generation process of software attacks as the manipulation of software failures, especially introduced by software crashes. We analyze the software crash by constructing a precise symbolic failure model, consisting of symbolic inputs, memory snapshots at the failure point (including concrete and symbolic values of all memory cells), and path constraints to reach the failure site. We propose a new automated exploit generation method based on $\mathrm{S^2E}$ with path selection optimization, selective symbolic input, and lazy evaluation on pseudo symbolic variables to handle symbolic pointers.

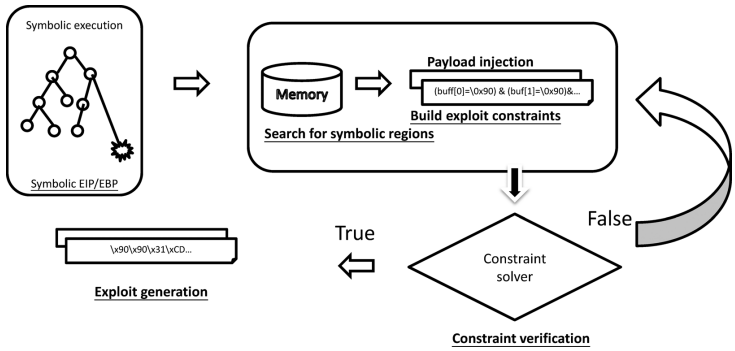

Fig. 1. The process of exploit generation from concolic-mode simulation with fuzz testing.

Concolic-mode simulation explores the failure path directly, and code selection filters complicated and unrelated library functions that do not affect exploit generation to reduce the overhead of symbolic executions. We view the concolic path exploring as the failure model construction process. During this process, CRAX also monitors the necessary conditions for exploit generation, and triggers an exploit generation process when it detects the condition. If the process fails to generate exploits, CRAX will go back and continue concolic execution.

A. Building the Failure Model

Software failure can be modeled as a symbolic execution trace, constituted of a set of symbolic inputs, memory snapshots at the failure point, and path constraints to reach the failure site. Given a crash input, we build the failure model by exploring the path introduced by the crash input. If an input data crashes a program, the execution path introduced by the crash input is very likely exploitable. Exploring the suspicious path is more effective than searching all paths.

1) Concolic-Mode Simulation: Concolic testing is symbolic execution, but it only explores the path introduced by concrete input. When a branch is encountered, only the direction in the concrete path will be explored, and path constraints will be updated so that a symbolic path will also be confined to the same direction at the branch.

When in symbolic execution, $\mathrm{S^2E}$ will use a constraint solver to determine the feasibility of each branch direction under current path constraints. To perform concolic execution, $\mathrm{S^2E}$ must follow only the direction introduced by concrete input. To implement this feature, we keep an extra constraint set, called *input constraints*. We replace path constraints by input constraints when $\mathrm{S^2E}$ is determining branch direction, and swap path constraints back when the branch direction is determined.

Input constraints are a constraint set that restricts every symbolic variable to a concrete value. Under the input constraints, every symbolic branch condition has only one possible value, so replacing path constraints by input constraints ensures $\mathrm{S^2E}$ follows only the concrete path. During the model construction process, we resolve the constraints by concrete value substitutions.

With fuzzer tools to identify an input crashing the program under test, concolic-mode simulation determines whether the path is exploitable rapidly because it focuses on only one path. Combining fuzzer tools with concolic-mode simulation provides a powerful technique for exploit generation as illustrated in Fig. 1.

2) Handling Symbolic Reads: Access violation is a common cause of a software crash, and is often due to read from or write to symbolic pointers. Handling symbolic pointers properly enables us to deal with more types of software crashes, and increase the opportunity of exploit generation. For symbolic writes, we treat them as a condition to trigger exploit generation, and will discuss this treatment in the following subsection. For symbolic reads, both solutions in $\mathrm{S^2E}$ and MAYHEM [16] are not suitable for symbolic pointers with large possible address ranges, which are common in corrupted pointers. So we also proposed a novel technique to handle symbolic reads with large possible address ranges.

The value read from a symbolic pointer can also be treated as a symbolic variable because we can change its value by manipulating the pointer. Therefore, we create a new symbolic variable to replace the result from each symbolic read, called a pseudo symbolic variable. Because the pseudo symbolic variables are unconstrained, we must put some constraints on them so that unreasonable paths will not be explored during the path exploration process. We will discuss two cases: concolic mode, and symbolic mode.

In the concolic mode, we use input constraints to determine the branch direction. When we create a new pseudo symbolic variable, we must add a corresponding input constraint so that only the concrete path will be explored.

The added constraint just restricts the pseudo variable to the value it should be in the concrete execution; that is, the pseudo variable is restricted to the value stored in the concrete address of the symbolic pointer (and if the value is still symbolic, get its concrete value again). The concrete value of a symbolic expression can be obtained using the SMT solver used by $\mathrm{S^2E}$ and the input constraints.

However, if a software crash is due to the access from a corrupted pointer, the concrete path usually ends with a segmentation fault, and no opportunity for exploit generation. Therefore, when a symbolic read with an illegal concrete address is detected, CRAX will switch to symbolic mode, try to continue execution, and wait for future opportunities for exploit generation. We have modified the solver for constraint reasoning with pseudo symbolic variables. The detail is in Section IV.

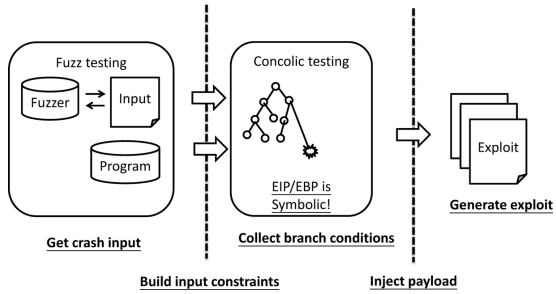

Fig. 2. The process of the exploit generation.

B. Necessary Conditions for Triggering Exploit Generation

1) Symbolic Program Counter (Symbolic EIP in x86 Machines): Because the EIP register contains the address of the next instruction to be executed, to control the register is the final target of all control-hijacking attacks. Thus, monitoring the state of the EIP register is a comprehensive, easy way to tackle different kinds of control-flow hijacking vulnerabilities. The process of detection of the symbolic EIP register and exploit generation is shown in Fig. 2.

2) Symbolic Write With Symbolic Data: In addition to the EIP register, corrupted pointers may change the control flow indirectly. Particularly, symbolic data assigned to a symbolic pointer means that arbitrary data can be written to arbitrary addresses. When a symbolic write is detected, the target of the writing operation will be redirected to sensitive data, such as return addresses, .dtors section, and GOT to update the EIP register indirectly. Considering Listing 1, an off-by-one overflow vulnerability will corrupt the *ptr* pointer, and as a result the value of *array[0]* may write to arbitrary addresses. Even if this vulnerability cannot corrupt return addresses directly, the symbolic pointer can taint the EIP register indirectly and hijack the control of a program.

Listing 1. An example code for pointer corruption

```
void test(int *input)
{
    int *ptr = array;
    int array[10];
    int i;
    for(i = 0 ; i <= 10; i++)
        array[i] = *(input + i);
    *ptr = array[0];
}
```

3) Symbolic Format Strings: The effect of a symbolic format string is similar to symbolic writes with symbolic data. With a carefully crafted format string, we can also write arbitrary data to an arbitrary address. CRAX also triggers exploit generation when a symbolic format string is detected. The detail is in Section V.

C. Exploit Generation

Given the failure model, the exploit generation process will search a variable assignment of symbolic variables that satisfy path constraints, and could redirect the control flow to the supplied shellcode.

1) Shellcode Injection: To inject the shellcode, we must find all memory blocks that are symbolic and large enough to hold the payload. Even if a symbolic block consists of many different variables, it could still be used to inject a shellcode as long as the block is contiguous. However, it is difficult to analyze source code manually to find a contiguous memory region that is tainted by user input and combined with variables. In addition, because the compiler often changes the order or the allocated size of variables for optimization, it is difficult to find a shellcode buffer manually. We automate this process by searching the maximum contiguous symbolic memory systematically.

2) NOP Sled and Exploit Generation: When the location of the shellcode is determined, *NOP sled* will try to insert a sequence of *NOP* instructions in front of the shellcode. This padding helps exploits against the inaccurate position of shellcode among different systems, or to extend the entry point of shellcode. Finally, the EIP register corrupted by symbolic data will point to the middle of NOP padding. All exploit constraints, including shellcode, NOP sled, and EIP register constraints, are passed to an SMT solver with path conditions to determine whether the exploit is feasible or not. If it is not feasible, the exploit generation goes back to the step of shellcode injection to change the location of shellcode until an exploit being generated or no more usable symbolic buffer.

D. Optimizations

1) Code Selection: Because S^2E performs symbolic execution on the entire operating system, the large number of path constraints will become an issue when the symbolic data are passed to the library or kernel. The constraints induced by the library or kernel are usually complex and huge, and constraint solvers often get stuck trying to solve them. For example, if the first argument of the fopen() function, which is a path of the file to be opened, is symbolic, then the constraint solvers will get a time-out error or hang in S^2E. Those paths in the library or kernel are often irrelevant to exploit generation. To avoid exploring those irrelevant paths, those library functions should run on concrete execution.

One of the essential features of S^2E is selective symbolic execution, which enables us to specify the region that should be

concrete executed. We can concretely execute library and kernel functions, and no path constraints will be added during the concrete execution. When this irrelevant function finishes, we can switch back to symbolic execution. S^2E will handle the switch between the concrete and symbolic values of each variable.

2) Input Selection: The size of the symbolic execution trace of the failure model is influenced by the number of symbolic input variables, the number of branch conditions carrying symbolic expressions, and the data flow to the failure event. Reducing the amount of symbolic input variables will also reduce the size of the symbolic execution trace. According to the observation, not all of the input will take part in the event manipulating process. That is, we can selectively use part of the input to speed up the failure model reconstruction. It is similar to unique pattern generations used by metasploit [22], which divert limited bytes of tainted inputs into the instruction pointer. Taintscope [23] uses a similar concept, called the selected inputs, as hot bytes to identify significant bytes for potential fuzzing operations, such as mutation, generation, or symbolic solution finding. If we can find the significant input bytes that will directly or indirectly (either consecutively or not) influence the continuations, we will be able to mark these inputs as symbolic. In metasploit's unique pattern generator, it is quite tricky to generate identifiable string patterns that taint the EIP or other registers. In S^2E with the proposed fast concolic method, we can partition the input into several smaller parts (100 byte units), making these parts symbolic individually. The hot bytes will be identified if the continuations are detected as symbolic.

III. Basic Implementation

The exploit generation steps are: 1) collect necessary runtime information, 2) build exploit constraints, and 3) pass the exploit constraints to the constraint solver to produce an exploit. The memory model in S^2E is also an important key to implementing the proposed methods. In addition to return-to-memory exploits, we also implement two other types of exploits, return-to-libc and jump-to-register exploits, to bypass some protections so that the exploit generation can be useful in real-world systems.

A. Symbolic Environment and Concrete Address Mapped Symbolic Memory

Symbolic environment is the primary way to attain the end-to-end capability. Concrete address mapped symbolic memory is the key for exploit generation on binary programs. Without full symbolic environment support, users must modify the source code to declare and make the input symbolic. S^2E is a whole system symbolic emulator, and all kinds of environment inputs in the operating system can be declared as symbolic including device inputs, network packets, sockets, files (including stdin), environment variables, and command arguments. The cost to take advantage of this feature is low. We use pipes to emulate symbolic *stdin*, and *mmap* to emulate symbolic files; all other environments can be easily emulated. With concrete address mapped symbolic memory, we can index symbolic memory by concrete addresses. Ordinary symbolic execution engines including KLEE, CUTE [24], DART [25], and CREST [26] can only index symbolic memory by abstract addresses. A practical exploit generation process is to search a usable address range to divert the failure event. Without concrete address mapped symbolic memory, we cannot analyze binary programs. The source code-based AEG is not sound, and must have an exploit validation process to revise the exploit address range.

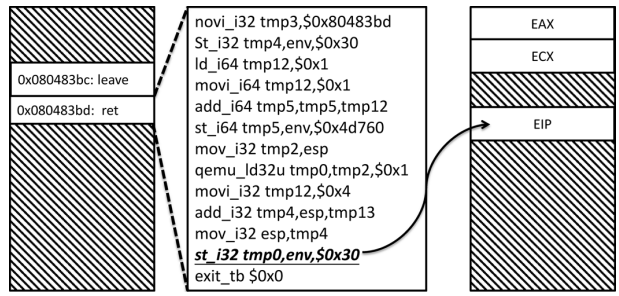

Fig. 3. The process of translating the *ret* instruction in QEMU.

B. Detection of Continuation Based Symbolic Registers

In QEMU, the *CPUX86State* structure is used to simulate the states of the x86 CPU, and all register references in a guest operating system will be turned into memory references on this structure. When S^2E is started, this structure is divided into two parts stored separately: *CpuRegistersState*, and *CpuSystemState*. The CpuRegistersState is a symbolic area which stores all the data in front of the EIP register in the CPUX86State structure, such as general-purpose registers. The CpuSystemState part is a concrete-only area that stores the other data including the EIP register.

S^2E translates every guest instruction into TCG IRs, and then translates those TCG IRs into host instructions or LLVM IRs. For instance, the *ret* instruction is separated into more detailed operations as shown in Fig. 3, and the operation of updating the EIP register is converted to a *store* instruction. In QEMU, all memory access operations are handled by a *softmmu* model to map the guest addresses to host addresses. Whenever accessing memory data, S^2E checks whether the value of the data point is symbolic or not in the softmmu model. If the value is symbolic, S^2E will rerun this translated block from the current instruction in KLEE to perform symbolic execution. To detect EIP register corruption, S^2E must check whether the writing target is the location of the EIP register, and whether the source value is symbolic whenever KLEE performs a *store* memory operation on symbolic execution.

When the EIP register is updated by symbolic data, the expression of symbolic data must be recorded because it describes which variable and which part of the symbolic data will update the EIP register. For example, given an expression that represents 32-bit symbolic data at the first element of an array named *buf* denoted as

$$(ReadLSB\ w32\ 0\ buf),$$

we can build a constraint to control the value of symbolic data, e.g., a constraint limiting the 32-bit data to zero as shown by

$$(Eq\ 0\ (ReadLSB\ w32\ 0\ buf)).$$

The current continuation has been set by the data, with constraints described by symbolic expressions. Next, we inject the shellcode into memory, and determine where the EIP register should point.

C. Exploit Generation

1) Memory Model in S^2E*:* In S^2E, the memory consists of *MemoryObject* objects, and the actual contents of these objects are stored in *ObjectState* objects. In an object of *ObejectState*, symbolic data are stored separately from concrete data. The expressions of symbolic data are stored in an array that consists of *Expr* objects, with pointers named *knownSymbolics* pointing to them. The concrete data are stored in an array of *uint8_t*, and pointed by a pointer named *concreteStore*. In each *ObejectState* object, a *BitArray* object named *concreteMask* is used to record the state of each byte, i.e., the byte is concrete or symbolic.

2) Finding Symbolic Memory Blocks: The default size of the storage in an *ObjecetState* object is 128 bytes. To find contiguous symbolic data in a memory region, we check the value of *concreteMask* structures sequentially, object by object. An object can be skipped easily whenever the values of its *concreteMask* structure are all ones; otherwise, the locations of every zero in the *concreteMask* structure must be recorded to compute the continuous size. For the symbolic blocks crossing different objects, it is necessary to check whether the current symbolic block is connected with the last symbolic block in the last object checked. The above procedure is shown in Algorithm 1.

Algorithm 1: Searching for symbolic blocks

Input: *Objects* : All *ObjectState* objects to be searched.

Output: V : A set of address and size.

```
1 foreach obj ∈ Objects do
    2 if isNotAllConcrete() then
        3 size ← 0
        4 for i ← 0 to 127 do
            5 if isByteSymbolic(i) then
                6 size ← size + 1
            7 else if size ≠ 0 then
                8 address ← getAddress(obj, i)
                9 if V → isConnect(address, size) then
                    10 V → updateLastItem(size); /⋆ A
                    part of the last block ⋆/
                11 else
                    12 V → addNewItem(address, size)
                    /⋆ An independent block ⋆/
                13 size ← 0
```

It is also required to determine the search range of memory regions. In Linux memory layout, the stack starts from the top at address 0xbfffffff and grows downward. It is easy to search the stack region from this address downward, but the heap and data segments are not necessarily located at a fixed address for different programs. Therefore, those starting locations need to be obtained dynamically. According to the ELF executable layout, the top of the executable files is the program header, which records all segment information. We can get the location and size of the data segment by analyzing the program header, which will be loaded to the address of 0x08048000.

On the other hand, because the heap region is behind the data segment and grows upward, the base address of the heap can be obtained by adding the starting address and size of the data segment.

3) Shellcode Injection: To determine whether shellcode can be stored in the potential buffers found by the previous step, each symbolic expression of a symbolic block needs to be read to build constraints that restrict each byte of symbolic data to a byte of shellcode sequentially byte by byte. This is an example showing the constraints that inject the shellcode into an array named *buf*:

$$\begin{gathered}(Eq\ 31\ (Read\ w8\ 0\ buf))\\(Eq\ C0\ (Read\ w8\ 1\ buf))\\(Eq\ 89\ (Read\ w8\ 2\ buf))\\(Eq\ C2\ (Read\ w8\ 3\ buf))\\\vdots\qquad\qquad .\end{gathered}$$

Next, the shellcode constraints are passed to an SMT solver with path conditions to validate their feasibility.

The best location for the shellcode is selected by having the NOP sled as large as possible. Therefore, all the symbolic blocks are sorted by size, and the shellcode is first injected from the end of the largest symbolic block. In addition to building the shellcode constraints, a new constraint needs to be added to ensure the EIP register can point to a range between the starting address of the shellcode and the top of the symbolic block. Even if the EIP register cannot point to the starting location of the shellcode precisely, it may be feasible because the NOP sled will extend the entry point later. If all of those constraints are infeasible, the location of the shellcode injection is shifted by one byte forward to try a new location iteratively.

In addition, shellcode will keep being injected into the current block or next blocks when those sizes are larger than the sum of the shellcode size and the current longest NOP sled size. For example, consider Fig. 4; the sum of the shellcode size and current NOP size is X, but it is smaller than Y and Z, so the shellcode and NOP sled will keep being injected into the next blocks and the current block. The algorithm is shown in Algorithm 2.

Algorithm 2: Injecting shellcode

Input: V : A set of address, and size of symbolic blocks. *Shellcode* : A shellcode string. *PC* : Path conditions.

Output: *ShellcodeAddress* : The starting location of shellcode injection. *MaxNopSize* : The max size of NOP sled.

```
1 foreach v ∈ V do
    2 if size_v ≥ strlen(Shellcode) then
```

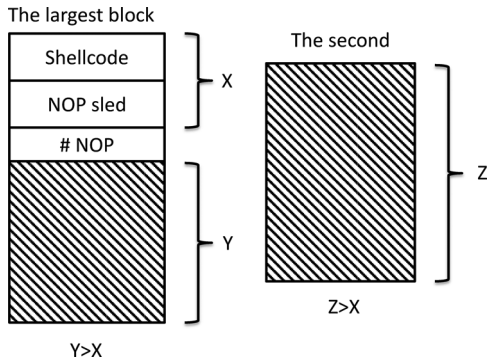

Fig. 4. The process of searching symbolic blocks.

3 address $\leftarrow$ address$_v$ + size$_v$ − strlen(Shellcode)

4 MaxNopSize $\leftarrow -1$

5 **while** $address \geq address_v$ **do**

6 c1 $\leftarrow$ injectShellcodeAt(address) /$\star$ Build shellcode constraints $\star$/

7 c2 $\leftarrow$ eipBetween(address, address$_v$) /$\star$ Build eip constraints $\star$/

8 **if** $Verify(PC \wedge c1 \wedge c2)$ **then**

9 nopSize $\leftarrow$ NOPSled(address, address$_v$, PC)

10 **if** $nopSize > MaxNopsize$ **then**

11 MaxNopSize $\leftarrow$ nopSize

12 ShellcodeAddress $\leftarrow$ address

13 **if**

$(address - address_v - nopSize) > strlen(shellcode) + MaxNopSize)$

then

14 address $\leftarrow$ address − nopSize

15 **else**

16 break

17 **else**

18 address $\leftarrow$ address − 1

4) NOP Sled: NOP sled aims to generate the more reliable exploits that increase the chance of success. CRAX will insert NOP instructions in front of the shellcode, as many as possible, and adjust the EIP register within the range. For efficiency, a binary search-like algorithm is used to determine the longest length of NOP sled rather than insert NOP instructions byte by byte. Whenever the binary search finds a range that the EIP register can point to, NOP instructions will be tried to fill this range sequentially to check whether both conditions are feasible simultaneously. If it is infeasible, the range is reduced; otherwise, the range is extended. The detailed algorithm is shown in Algorithm 3.

Algorithm 3: NOP sled

Input: *Start* : The starting address of NOP sled. *End* : The end address of NOP sled. *PC* : Path conditions.

Output: *NopSize* : The size of NOP sled.

1 min $\leftarrow$ End

2 max $\leftarrow$ Start

3 mid $\leftarrow$ min +(max − min)/2

4 **while** min $\leq$ max **do**

5 c1 $\rightarrow$ eipBetween(Start, mid) /$\star$ Build eip constraints $\star$/

6 **if** $Verify(PC \wedge c1)$ **then**

7 c2 $\rightarrow$ injectNopBetween(Start, mid) /$\star$ Build NOP constraints $\star$/

8 **if** $Verify(PC \wedge c2)$ **then**

9 NopSize $\leftarrow$ Start − mid

10 max $\leftarrow$ mid − 1

11 **else**

12 min $\leftarrow$ mid + 1

13 **else**

14 max $\leftarrow$ mid − 1

15 mid $\leftarrow$ min +(max − min)/2

After the longest length of the NOP sled is obtained, the next step is to make the EIP register point close to the middle of the NOP sled. Because the number of NOP sleds may be large, the constraint solver is used to reason out the suitable location where the EIP register points. To help a constraint solver to compute the address as close to the middle of the NOP sled as possible, a constraint is added to limit the range. First, the range is a point in the middle of the NOP sled, and the constraints are passed to a constraint solver to get the solution. If it is infeasible, the range is extended twice larger each time, and so on. This process can obtain a solution because the previous step guarantees that the EIP register can point to the range of the NOP sled. The algorithm is shown in Algorithm 4.

Algorithm 4: Determining the value of the EIP register.

Input: *NopSize* : The size of NOP sled. *Start* : The start address of shellcode. *PC* : Path conditions.

Output: *EipAddress* : The address where EIP register points at.

1 mid $\leftarrow$ Start − (NopSize/2)

2 range $\leftarrow$ 0

3 **repeat**

4 **if** $mid - range \leq Start - NopSize$ **then**

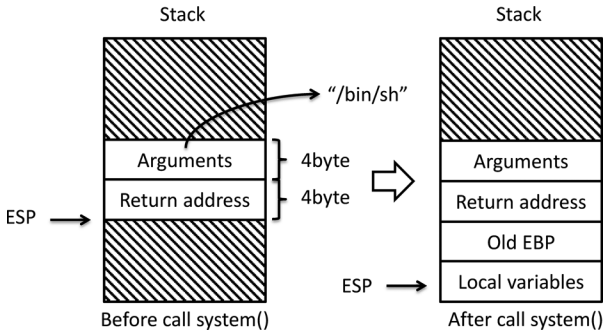

Fig. 5. The process of return-to-libc exploit generation.

5 low ← Start − NopSize

6 **else**

7 low ← mid − range

8 **if** $mid + range \geq Start$ **then**

9 hi ← Start

10 **else**

11 hi ← mid + range

12 c ← eipBetween(low, hi) /⋆ Build eip constraints ⋆/

13 **if** $range = 0$ **then**

14 range ← 1

15 **else**

16 range ← range ∗ 2

17 **until** $Verify(PC \wedge c)$;

18 EipAddress ← getValue(PC ∧ c)

Finally, the starting address of the shellcode, the size of the NOP sled, and the location EIP register points are to be determined if they are feasible. The constraint solver will solve the final path conditions to generate the exploit that performs the malicious task in the shellcode.

D. Other Types of Exploit

1) Return-to-Libc: A return-to-libc attack is a technique to bypass non-executable memory regions, such as W ⊕ X protection. It redirects control flow to functions in the C runtime library, such as system(), and injects the arguments of the function into the stack manually to fake the behavior of function callers. Because the runtime library is always executable and loaded by operating systems, a return-to-libc attack can perform malicious tasks by executing library code, and bypass the executable space protection. Fig. 5 shows the process of calling the system() function of libc. The argument containing the command string will be pushed into the stack. It does not matter where the libc function call returns, but the arguments are the key to perform the tasks in which we are interested.

Taking system("/bin/sh") as an example, which will open a shell, the only argument is a pointer that points to the string "/bin/sh" as shown in Fig. 5. The process of return-to-libc exploit generation is similar to return-to-memory. We also need to inject shellcode and NOP sled, just with different contents, as shown in Table I.

TABLE I
THE DIFFERENCES BETWEEN RETURN-TO-MEMORY AND RETURN-TO-LIBC EXPLOIT

Exploit	Shellcode Injection	NOP Sled
Return-to-memory	shellcode	NOP instruction(0x90)
Return-to-libc	"/bin/sh"	whitespace(0x20)

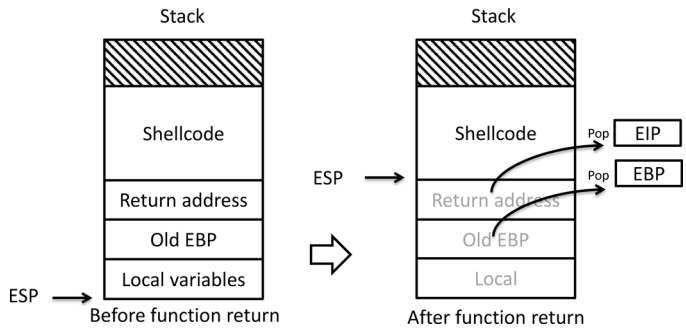

Fig. 6. The process of jump-to-register exploit generation.

Shellcode injection injects the string "/bin/sh" instead of a shellcode, and NOP sled fills the stack with white space characters rather than NOP instructions.

2) Jump-to-Register: The stack is the most common memory region for shellcode injection, but ASLR randomizes the base address of the stack so that control flow does not jump to shellcode accurately. A large NOP sled may bypass ALSR, but it is not always feasible. A jump-to-register attack is a technique to bypass ASLR. It uses a register that points to a shellcode as a trampoline to execute the malicious tasks. For example, the Extended Accumulator Register (EAX) is usually used to store the return value of functions. The strcpy() function returns a pointer that points to the location of buffer, and the EAX is often used to store the address. If a "call %eax" instruction can be found in the code segment, and shellcode can be injected into the buffer that the EAX register points to, then flow control will be redirected to execute this instruction and jump to shellcode.

In addition, jump-to-esp is also a common, reliable attacking technique which doesn't need to insert NOP sled and guess the stack offset in Windows and old versions of Linux. When a function returns its results, the return address will be popped, and the ESP register will point to the stack entry next to the entry that stores the return address. We can inject shellcode behind the return address, and use the ESP register as a trampoline. If a "jmp %esp" instruction can be found in the code segment, a jump-to-esp exploit can be generated to bypass the ASLR. The process is shown in Fig. 6.

To generate jump-to-register exploits, a code segment must be searched to find the related instructions such as "call %eax" and "jmp %esp". If the related instructions are found, and the memory region that the register points to is symbolic, shellcode will be injected into the location, and the EIP register will be redirected to execute the related instruction. In addition, if there

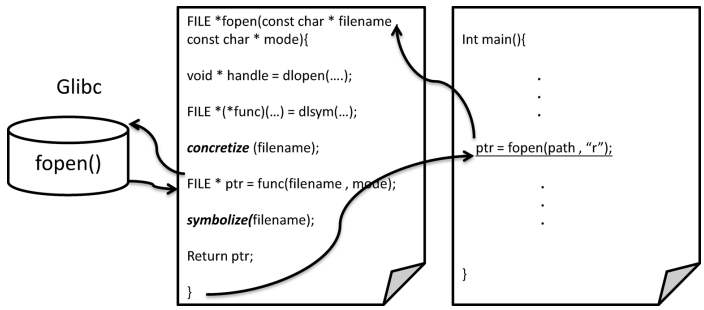

Fig. 7. An example for code selection.

is not any usable instructions in the code segment, the data segment may be searched to find two-byte symbolic data to inject into the related instruction because the data segment is unaffected by ASLR. For example, the "jmp %esp" instruction is 0xffe4 and the "call %eax" instruction is 0xffd0.

E. Concolic-Mode Simulation

To simulate concolic testing on S^2E, we execute the program under test with input data, such as arguments and environment variables, while the main tasks are building input constraints and collecting branch conditions. According to the memory model in S^2E, the concrete values are stored separately from symbolic data, but the concrete values are ignored because the variables are marked as symbolic. To build input constraints, we must obtain concrete input data at run time, which can be done by reading the last concrete value of the symbolic variables from the *concreteStore* structure. A vector container is used to save all input constraints because it is easy to delete some constraints when they are unnecessary, and to combine every constraint into a complete input constraint when it is needed.

In symbolic execution, if a symbolic branch condition is only feasible in one direction, there is no need to add the branch condition to path constraints. In concolic execution, we must collect every symbolic branch conditions into path constraints because we only follow the concrete path, and never use the solver to determine the feasibility of each branch direction.

In addition to branches, symbolic addresses also cause state forks on symbolic execution. When accessing a memory address whose value is symbolic, S^2E cannot determine where it should access. S^2E forks executions to try to access every address to which the symbolic address can refer. In concolic mode, input constraints are still used to help SMT solvers determine a location on a symbolic address.

It is easy to switch between concolic-mode simulation and original symbolic execution because the memory model of S^2E is not modified. If symbolic execution will be performed, we just set the input constraints to be true because path conditions will not be changed when they perform an AND logical operation with a true expression.

F. Detection of Pointer Corruption

When accessing a memory address, S^2E will check whether the address is symbolic or not. If it is symbolic, S^2E must determine an explicit location before keeping program execution. S^2E uses a binary search to find all locations that the symbolic address can point to, and forks executions to explore each address. Before S^2E handles a symbolic address, the address can try to point to sensitive data, such as a return address or a GOT. If it is feasible, those sensitive data will be tainted and corrupt the EIP register later. Otherwise, the symbolic address will be changed to taint other concrete data because it may help exploit generation if other vulnerabilities corrupt the EIP register later. For example, we can change the address to taint the data segment so that the shellcode can be injected to bypass ASLR protection.

G. Code Selection

Because S^2E has built-in selective symbolic execution, it is easy to use this capability to select the code to run on concrete execution or symbolic execution. In Linux, the LD_PRELOAD environment variable can intercept the library functions, and jump to the functions. With the help of this environment variable, we can intercept those irrelevant library functions and concretely execute them. Fig. 7 shows an example that intercepts the *fopen* function and performs concrete execution on it. In addition, some functions just print messages to the screen without a return value, such as *perror()*, so those functions can be skipped using this method directly to speed the process of exploit generation.

IV. Reasoning With Pseudo Variables

Vulnerabilities that involve corrupted pointers, like heap overflow, cannot be exploited without a proper symbolic pointer handling mechanism. However, mechanisms in S^2E and MAYHEM [16] are not suitable for corrupted pointers because they can only handle symbolic pointers with small address ranges. Hence, we introduce a new approach that enables CRAX to handle symbolic pointers with large address ranges. We have introduced pseudo symbolic variables and lazy evaluation for resolving possible pointer values for exploit generation.

A. Pseudo Symbolic Variable for Memory Management Helpers

We modified the memory management helpers of S^2E. When symbolic read occurs, we first check whether the symbolic pointer can refer to a legal address under the current path constraints. If not, we concretize the pointer. If the symbolic pointer can refer to a legal address, we then create a new symbolic variable (called the pseudo symbolic variable) to

replace the read result and record all information needed for future constraint solving.

Information from the symbolic read is packed into a data object named Dereference, in which we have the fields Dereference.address, Dereference.value, and Dereference.memory corresponding to the symbolic address expression, the pseudo symbolic variable created for this read, and the memory snapshot at the moment that the symbolic read occurs, respectively. For each state, we maintain a set D that contains all Dereference objects created during the execution of the state. Newly created Dereference objects will be stored in the set.

For the memory snapshot, rather than scanning all possible addresses, we instrument $\mathrm{S^2E}$ to fork a state (called meta state), and remove the meta state from the execution queue. $\mathrm{S^2E}$ has a built-in copy-on-write mechanism to keep track of the memory content of every state. Because the meta state has never been executed, we can access the memory content (using the readMemory method) at the moment when the meta state is forked.

B. Lazy Evaluations of Pseudo Symbolic Variables by Late Address Assignment

We treat the result of a symbolic read as a newly created, un-constrained pseudo symbolic variable. Because the value read is determined by the address, if we pass an expression that contains pseudo symbolic variables to the constraint solver, we will likely obtain an infeasible answer. We implement a modified version of the constraint solver routines so that the consistency of symbolic reads can also be ensured.

The following are typical constraint solver routines used by $\mathrm{S^2E}$, where $bExpr$ is a Boolean expression.

1) $mayBeTrue(bExpr)$: Check if there is some variable assignment that can make $bExpr$ become true.
2) $mayBeFalse(bExpr)$: Check if there is some variable assignment that can make $bExpr$ become false.
3) $mustBeTrue(bExpr)$: Check if $bExpr$ is provably true (no variable assignment can make it false).
4) $mustBeFalse(bExpr)$: Check if $bExpr$ is provably false (no variable assignment can make it true).

Every routine can be implemented by any other one. For example, $mayBeTrue(bExpr)$ can be implemented as $!mustBeFalse(bExpr)$. Therefore, we only implement a modified version of $mayBeTrue$, and the other three can be easily implemented. In the following discussions, we use $oMayBeTrue$ to denote the original solver routine, and $mMayBeTrue$ to denote the modified version. The other three routines are denoted similarly.

For the $mMayBeTrue(bExpr)$ routine, we must determine whether we can find a variable assignment (including the assignment to pseudo symbolic variables) that makes $bExpr$ true, and also preserve the consistency of all symbolic reads. More formally, if V is a variable assignment, and e is a symbolic expression (Boolean or numerical), we define $V(e)$ to be the value of e under the variable assignment V. The goal of $mMayBeTrue(bExpr)$ is to determine whether there exists a variable assignment V that satisfies

$$V(bExpr) \wedge \left(\bigwedge_{d_i \in D} V(d_i.value) = V(d_i.memory[V(d_i.address)]) \right). \tag{1}$$

We can reuse the power of the original solver routine by transforming the problem into determine whether there exists a mapping A that maps each element d_i of D to a concrete address $A(d_i)$, and satisfies equation (2) at the bottom of the page.

Note that $d_i.memory[A(d_i)]$ may also be a symbolic expression.

Determining the satisfiability of (1) is equivalent to determining that of (2). If (2) can be satisfied, we must have a variable assignment V that satisfies equation (3) at the bottom of the page.

This condition is what $oMayBeTrue$ is designed to answer. Obviously the variable assignment V also satisfies (1). In the reverse direction, if we can find a variable assignment V that satisfies (1), then we take the mapping

$$A(d_i) = V(d_i.address).$$

The mapping A also satisfies (2).

We define the address assignment of D as a mapping that maps every element of D to a concrete address. An address assignment A of D satisfies $bExpr$ if (2) is true. We extend these definitions to the subset D' of D by replacing D with D' in the above definitions (and also (2)). We also define the address assignment of a single $Dereference$ object to be the concrete address that will be mapped from the object.

$$oMayBeTrue \left(bExpr \wedge \left(\bigwedge_{d_i \in D} d_i.address = A(d_i) \wedge d_i.value = d_i.memory[A(d_i)] \right) \right). \tag{2}$$

$$V(bExpr) \wedge \left(\bigwedge_{d_i \in D} V(d_i.address) = A(d_i) \wedge V(d_i.value) = V(d_i.memory[A(d_i)]) \right). \tag{3}$$

The goal of $mMayBeTrue(bExpr)$ is to find an address assignment of D that satisfies $bExpr$, or determine that such an assignment does not exist.

1) Searching Address Assignment of a Singleton Subset: We must find an address assignment of a singleton subset d of D that satisfies $bExpr$. Because the address assignment only contains one concrete address, we use binary search for every possible concrete address, and use solver to determine the satisfiability of bExpr under each assignment. The process is shown in Algorithm 5. However, we still need optimization techniques for general cases.

Algorithm 5: FindSatisfyingAddress

Input: *bExpr* : The constraint. *d* : The dereference object. *addrMin* : The starting address. *addrMax* : The end address.

Output: *addr* : The address satisfies the constraint.

1 **if** *The address range[addrMin, addrMax] is unmapped* **then**

2 return null

3 **if**

$\neg oMayBeTrue(bExpr \wedge d.address \geq addrMin \wedge d.address \leq addrMax)$

then

4 return null

5 **if** $addrMin = addrMax$ **then**

6 **if**

$oMayBeTrue(bExpr \wedge d.address = addrMin \wedge d.value = d.memory[addrMin])$

then

7 return addrMin

8 return null

9 **else**

10 addrMid $\leftarrow$ (addrMin + addrMax)/2

11

addr $\leftarrow$ FindSatisfyingAddress(bExpr, d, addrMin, addrMid)

12 **if** $addr \neq null$ **then**

13 return addr

14

addr $\leftarrow$ FindSatisfyingAddress(bExpr, d, addrMid+1, addrMax)

15 **if** $addr \neq null$ **then**

16 return addr

17 return null

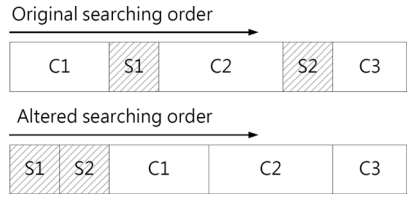

Fig. 8. Alter searching order; the original order is from the lower address to the higher address, and the altered order is searching symbolic blocks first.

Value Based Filtering: If $bExpr$ strictly constrains $d.value$ but with few constraints on $d.address$, we must perform almost 2^{32} solver queries to find a satisfying address assignment. However, if $d.value$ is constrained to a few concrete values, we can collect all the possible values of $d.value$ (using similar binary search technique). Before recursively searching an address range, we first check if any memory cell in the range contains possible values. If not, there is no need to search into the address range.

Currently, we will trigger this optimization when $d.value$ has fewer than 5 possible values, and the address range (that is, $addrMax - addrMin$) is smaller than 1024.

Searching Symbolic Blocks First: $bExpr$ may put no constraint on $d.value$ or $d.address$, but on their relationship, for example, $bExpr$ may look like $d.address + d.value = 0x12345678$. In this case, the optimization technique introduced previously is useless. To handle this case, we alter the searching order to search the symbolic blocks first, as Fig. 8 illustrated.

If we cannot find a satisfying address assignment from symbolic blocks, we will try the concrete blocks. However, this may take a long time, so we also impose a timeout value when searching concrete blocks, and judge that there is no satisfying address assignment if the timeout is reached.

The *FindNextSatisfyingAddress* function will find an address assignment of d that is greater than or equal to addr in the searching order. The *GetNextAddress* function will return the next address of addr in the searching order. We also implement a modified version of the two functions to follow the altered searching order.

2) Searching Address Assignment of D: The intuitive way to search an address assignment of D satisfying $bExpr$ is to select an element d of D, find a satisfying address assignment A of d, and recursively search the address assignment of the set $D - \{d\}$ that satisfies $bExpr \wedge d.address = A(d) \wedge d.value = d.memory[A(d)]$.

In most cases, the unsatisfiability of $bExpr$ can be judged by examining only a few key Dereference objects, but if those objects are examined in deep recursions, we would waste time backtracking, as illustrated in Fig. 9. In Fig. 9, each round corner rectangle stands for an initial or recursive call of *HasAssignment*, and arrays of squares are memory snapshots of each Dereference object passed by the second parameter of *HasAssignment*. The selected object d is denoted below

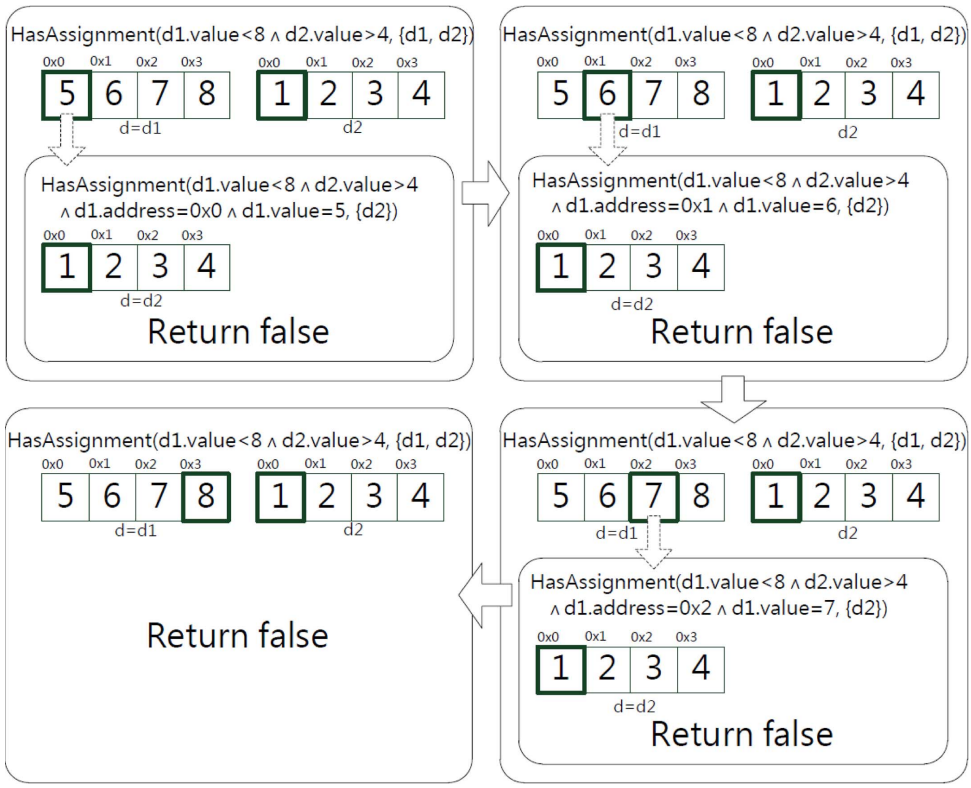

Fig. 9. The searching process of Algorithm 6 without re-selection of d in D.

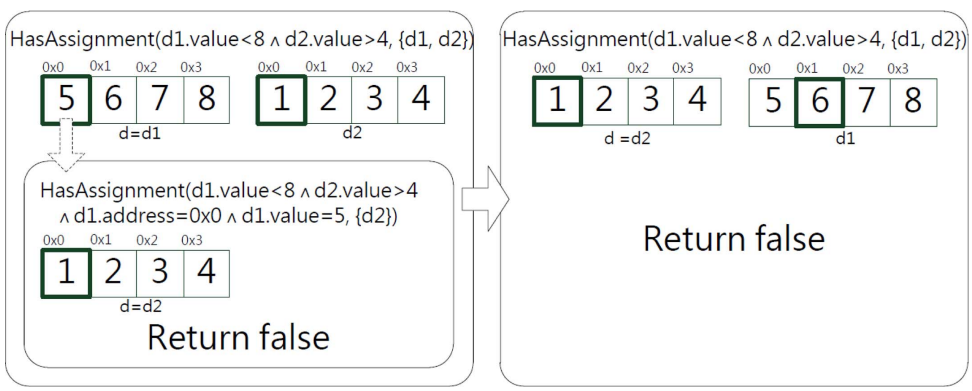

Fig. 10. The searching process of Algorithm 6.

the memory snapshot, and the memory cell pointed by the lastAddress field is denoted by a thick border. Although (at line 4 of Algorithm 6) we can use a smarter heuristic to select the Dereference object to solve, there is no guarantee that the heuristic always makes the best decision.

Therefore, we change the Dereference object we are solving. The selection heuristic at line 13 has one direction to follow: select the Dereference object that we just failed to find a satisfying assignment at the recursive call of line 10 because it is likely to be one of the key Dereference objects mentioned before. Fig. 10 illustrates the searching process of Algorithm 6 with the same example as Fig. 9; we can see that, after the second recursion fails, the first recursion re-selects a Dereference object, which is the one for which the second recursion just failed to find a satisfying assignment.

Additionally, we can prove that Algorithm 6 will eventually terminate by induction on the size of D; Algorithm 6 must terminate if $|D| = 0$. Assume Algorithm 6 will terminate if $|D| \leq k$; then when the $|D| = k + 1$, line 10 must terminate because $|D - \{d\}| = k$. If the recursive call at line 10 returns true, then Algorithm 6 terminates; otherwise, *d.lastAddr* will be incremented, so we will not try the same d with the same $d.lastAddr$ at line 5, which implies Algorithm 6 must terminate.

Algorithm 6: HasAssignment

Input: *bExpr* : The constraint. D : The Dereference object set.

Output: *Result* : if there exists an address assignment satisfying the constraint.

1 **if** D *is empty set* **then**

2 return true

3 Select an element d of D

4 **repeat**

5 addr $\leftarrow$ FindNextSatisfyingAddress(bExpr, d, d.lastAddr)

6 **if** $addr = null$ **then**

7 return false

8 d.lastAddr $\leftarrow$ addr

9

$$\text{constraint} \leftarrow \text{bExpr} \wedge \text{d.address} = \text{addr} \wedge \text{d.value} = \text{d.memory}[\text{addr}]$$

10 **if** $HasAssignment(constraint, D - \{d\})$ **then**

11 return true

12 d.lastAddr $\leftarrow$ GetNextAddress(d, d.lastAddr)

13 Re-select an element d of D

14 **until** *false*;

C. Exploiting the Unlink Process of Free()

With the symbolic pointer handling mechanism, we can automatically exploit the unlink macro used by free() in an early version of glibc. The following code is the unlink macro. When heap metadata are overwritten by symbolic data, P will be symbolic, and FD and BK will become the result of symbolic reads, so CRAX will produce two pseudo variables to replace the value of FD and BK. Because FD also becomes symbolic, line 5 will become a symbolic write, which allows us to redirect the GOT entry or return address to the shellcode.

Listing 2. The unlink macro

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {
    FD = P-> fd;
    BK = P-> bk;
    FD-> bk = BK;
    BK-> fd = FD;
}
```


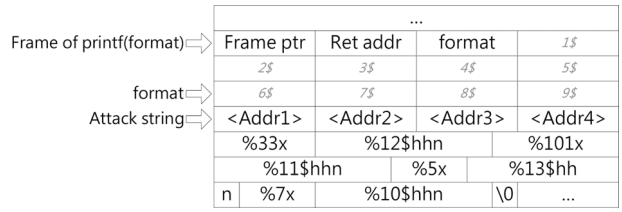

Fig. 11. The stack layout when attacking printf; italic text denotes the $-offset needed to access the cell.

However, there will be a second symbolic write at line 6, and the target is just 8 bytes after the shellcode address, which will corrupt the shellcode. To handle this condition, we add a jump instruction at the beginning of the shellcode to jump over the corrupted area, and we do not try to insert NOPs when we are exploiting heap overflow vulnerabilities.

V. Format String Exploit Generation

Format string vulnerability is a common vulnerability caused by misuse of formatting functions such as *printf* or *syslog*. Attackers can overwrite an arbitrary address with an arbitrary value by injecting a carefully crafted attack string into the format string. Fig. 11 demonstrates the common pattern of attack string used in CRAX's format string exploit. The first 4 addresses point to the location we want to overwrite, usually the GOT entry of some library function. Because only one byte would be overwritten for each %hhn formatting, we need 4 addresses to overwrite the 4-byte GOT entry. In the remaining part of the attack string, we repeatedly use the %x formatting to manipulate the length of the printed string, and the %hhn formatting to write the string length to the address specified in the beginning of the attack string.

There is another problem: we must ensure that each %hhn formatting can refer to one of the addresses at the beginning of the attack string. This referencing can be done by using the '$' formatting option, which assigns the argument with pointer values. As in Fig. 11, the number before the '$' formatting option (denoted as $-offset) specifies which argument to use. Because the arguments are passed by the stack, we can access any portion of the attack string (with carefully chosen $-offset) if the attack string is also located in the stack.

Currently, CRAX only supports format string exploit generation in Linux. To capture invocations of formatting functions in the guest OS, we use the LD_PRELOAD environment variable to pre-load the re-implementations of formatting functions. Once formatting function invocation is captured, the exploit generation process is triggered. The generation process of the format string exploit consists of several steps: 1) format string vulnerability detection, 2) $-offset detection, and 3) constraint reasoning and exploit generation.

A. Format String Vulnerability Detection

Once the formatting function call is detected, we first check whether this call is vulnerable. That is, we check whether the format string contains symbolic data. If it does, we continue the remaining exploit generation process; otherwise we redirect control flow to the original formatting function in libc as

nothing happens. Currently, we try to generate format string exploits only when the format string contains more than 50 bytes of symbolic data.

B. $-Offset Detection

If a vulnerable formatting function call is detected, then we will try to generate an exploit. The first information we must obtain is the $-offset that can reach the beginning of the format string. Although we do not have to put the attack string at the beginning of the format string, we must know the $-offset to reach the attack string, which can be computed by the $-offset to the beginning of the format string plus the distance between the formatting string and the attack string. Perhaps we can compute the $-offset directly by the distance between the current frame pointer and the location of the format string, but for different formatting functions, there may be variations in their $-offset so that we should compute it case by case. However, the offsets usually are small numbers, so we instead adopt a linear search method. We just try every natural number starting with 1, and test whether it is the correct $-offset.

Algorithm 7: Search$-offset

Input: *FormatString* : the format string.

Output: *Offset* : Searching $-offset.

```
1 Offset ← 1
2 M ← MagicNumber
3 repeat
    4 if stateFork() ≠ 0 then
        5 exitStatus ← waitState()
        6 if exitStatus = true then
            7 break
        8 else
            9 Offset ← Offset + 1
    10 else
        11 NormalTerminateOnMemoryAccess(M)
        12 FormatString ← GenTestString(M, Offset)
        13 OriginalFormattingFunction(FormatString)
        14 ErrorTerminate()
15 until false;
```

The pseudo code of how we discover the correct $-offset is shown in algorithm 7. We implement a set of POSIX-fork like API for a guest OS program to control the state fork in S^2E. When the pre-loaded library detects format string vulnerability, it will enter a loop continually trying each possible $-offset. The trials are done in the child state, and the parent state will be informed of successful trials by the exit status of the child state. The child state tests the given $-offset by replacing the head of the format string with a special test string, which is prefixed by a 4-byte magic number M, and followed by a %n formatting with the given $-offset. When printing the modified format string with correct $-offset, the %n formatting will reference the magic number M in the beginning of the test string, and cause a memory write to address M. Before printing the format string, the child state will use the API to instruct S^2E to capture such a memory write. If such a memory write is captured, the child state will be terminated, and the parent state will be informed of a successful trial.

C. Constraint Reasoning and Exploit Generation

Now that we have all the information needed for generating an attack string, the only thing left is inserting the shellcode and NOP sled into some symbolic memory blocks, and inserting an attack string to overwrite the pre-specified GOT entry into the address of the shellcode. Shellcode and NOP sled are discussed previously, and the role of the attack string is similar to the EIP constraint in the previous discussions. However, inserting an attack string is more complicated than manipulating a symbolic EIP. We don't have a dedicated symbolic EIP expression, but we have a symbolic formatting string. We have to generate an attack string. If the attack string is shorter than the formatting string, we can insert the attack string in different locations in the formatting string. Because we don't have the symbolic EIP expression, we cannot use the binary search mechanism to assign the EIP register to the address nearest to the middle of the NOP sled. Hence, we currently just iteratively try each possible location within the formatting string where we can put the attack string, and for each possible location we also try each possible address to which we can overwrite the GOT entry (we must generate different attack strings for different addresses), with the priority that the addresses near the middle of the NOP sled are tried first. Once a satisfactory setting is found, we can pass the shellcode, NOP sled, and attack string constraints to the constraint solver to produce an exploit.

VI. Experimental Results

We conducted five types of experiments to evaluate the work for automatic exploit generation. The first experiment is with five different common control-flow hijacking vulnerabilities. The experiment demonstrates that the proposed method can handle all vulnerabilities that symbolically update the EIP register, and some vulnerabilities that symbolically update pointers. The second experiment is with return-to-libc and jump-to-register exploit generations to demonstrate that the method could bypass some mitigation protections in real-world systems. In the third experiment, we generated exploits for 33 real-word programs, most chosen from the benchmark of AEG and MAYHEM, to demonstrate that our method can handle at least most of all cases that AEG and MAYHEM addresses. Although we are conducting the whole system symbolic execution and AEG is on the application level, our method is still faster than AEG because we reduce the number of constraints by performing concolic execution on selected code and input. The fourth experiment reveals the performance speedup between the original concolic method and the improved method. The speedup can achieve 7000 times faster execution than our initial

TABLE II
THE RESULTS OF EXPLOIT GENERATION FOR SAMPLE CODE

Vulnerability	Corrupted Data	Concolic Wall Time(sec.)	Symbolic Wall Time(sec.)
Stack buffer overflow	Return address	0.61/3.59	0.69/303.59
Heap buffer overflow	Pointer	0.24/3.16	0.25/301.73
Off-by-one overflow	EBP register	0.46/3.24	0.51/302.14
Uninitialized variable	Function pointer	0.41/3.59	0.46/303.23
Format string	Pointer	0.05/4.81	–
Average		0.35/3.67	0.47/302.67

TABLE III
THE RUN-TIME INFORMATION OF RERUN-TO-LIBC EXPLOIT GENERATION

Run-time Information	
EIP register	(ReadLSB w32 54 arg)
ESP register	0xbffff8e0 (value:(ReadLSB w32 58 arg))
ESP register + 4	0xbffff8e4 (value:(ReadLSB w32 62 arg))
Address of system()	0xb7ebb7a0
Potential shellcode buffers	0xbffffa8f (size:100 bytes) 0xbffff8a6 (size:100 bytes)

attempt, due to this performance tuning. We also demonstrate the power of CRAX to produce exploits of large applications, including *Microsoft office word*, *foxit pdf reader*, and *Mplayer*.

A. Testing Method and Environment

All of the experiments are performed on a 2.66 GHz Intel Core 2 Quad CPU with 4 GB of RAM, and the host environment is Ubuntu 10.10 64-bit. The guest environment is Debian 5.0 32-bit with default settings of QEMU, which is a 266 MHz Pentium II (Klamath) CPU with 128 MB of RAM.

Most of the programs under test are compiled by GCC 4.3.2, and run on Glibc 2.7, which are the default in Debian 5.0. The other programs use GCC 3.4.6 and Glibc 2.3.2 to generate exploits. The default version of GCC protects the main functions against stack buffer overflow, and performs heap integrity checks to stop heap overflow.

We use an end-to-end approach to generate exploits on binary executables without modifying the source code. Our approach is to fork a new process to execute the program under test, and pass the symbolic data to it from the outside. Symbolic data are created by the control process and passed to the target by interprocess communication methods. For example, the control process maps a buffer to a file by mmap(), and make this buffer as symbolic. Whenever the target program accesses the memory mapped file (with the corresponding memory as symbolic), S^2E will start symbolic executions. Other kinds of symbolic environment can be simulated in the same way.

In Debian 5.0, ASLR is enabled by default so that the based address of stack and heap is randomized. Therefore, ASLR is disabled in our experiments for generating and testing all exploits except jump-to-register exploits.

B. Sample Code

Because our method is based on detection of a symbolic EIP register, it can handle different types of vulnerabilities. In the first experiment, we design code examples for five different vulnerabilities and four types of corrupted data, and perform automated exploit generation on them. The results are shown in Table II, where the wall time is expressed by *(exploit reason time/total time)*.

In this experiment, the inputs of all sample codes are arguments, and the length of all inputs are 100 characters. We compare the efficiencies of concolic-mode simulation with traditional symbolic execution. In symbolic execution, depth-first search (DFS) is used to explore a symbolic execution tree. The heap overflow code is executed on Glibc 2.3.2 because some protections that check pointer consistency have been included in Glibc since version 2.3.6. In addition, this exploit generation cooperates with the *libfmtb* library to build format strings to exploit format string vulnerabilities.

In the five vulnerabilities considered in the first experiment, stack buffer overflow and uninitialized variable vulnerability corrupt the EIP register directly, and the other three vulnerabilities taint the EBP register or pointers to corrupt the EIP register indirectly. As the results show, the average total time is 3.67 seconds in concolic mode, and the exploit reason time is 0.35 seconds. On average, symbolic execution spent 302.67 seconds on generating an exploit, and 0.47 seconds on reasoning it out. Concolic mode was faster by about 100 times than symbolic execution because it just explored only one suspicious path. In the experiments of symbolic execution, format string vulnerability got an out-of-memory error because symbolic execution attempted to explore all paths in the *snprintf()* function, which performs a complex behavior.

C. Return-to-Libc, and Jump-to-Register Exploits

In the second experiment, we implemented return-to-libc and jump-to-register exploit generation, and the generated exploits could bypass non-executable stacks or ASLR protection.

Because return-to-libc and jump-to-register exploit generation do not apply to all cases, we choose the sample code of stack buffer overflow vulnerability to conduct the experiment. Furthermore, the experiment was conducted on the concolic mode.

In the experiment on return-to-libc exploits, we use the system() function to execute a "/bin/sh" command. The run-time information at exploit generation is shown in Table III. The top of the stack (which is the location that stores the argument of the system() function) is symbolic, so we can manipulate the argument (which is a pointer point to the command string) so that it points to our shellcode buffer. And the potential shellcode buffers were large enough to insert the string "/bin/sh". Therefore, this vulnerable program satisfied all the conditions for return-to-libc exploit generation. The time spent on the exploit reason was 0.34 seconds, while the total time of this experiment was 3.25 seconds.

Table IV shows the run-time information at jump-to-register exploit generation. A call %eax instruction was found at address 0x0804839f, and the EAX register pointed to the starting location of a symbolic region exactly. Therefore, this vulnerable

TABLE IV
THE RUN-TIME INFORMATION OF JUMP-TO-REGISTER EXPLOIT GENERATION

Run-time Information	
EIP register	(ReadLSB w32 54 arg)
Usable trampoline registers	EAX
EAX register	0xbffff8a6 (value:(ReadLSB w32 0 arg))
Address of "call %eax" instruction	0x0804839f
Potential shellcode buffers	0xbffffa8f (size:100) 0xbffff8a6 (size:100)

TABLE V
THE RESULTS OF EXPLOIT GENERATION FOR REAL-WORLD PROGRAMS WITHOUT SELECTIVE INPUT

Program	Version	Input Source	Input Length	Wall Time(sec.)	Advisory ID.
aeon	0.2a	Env. Var.	550	12.36/32.03	CVE-2005-1019
iwconfig	V.26	Arguments	85	0.89/3.57	BID-8901
glftpd	1.24	Arguments	300	2.68/8.06	OSVDB-16373
ncompress	4.2.4	Arguments	1050	45.57/99.36	CVE-2001-1413
htget	0.93	Arguments	276	8.21/35.48	CVE-2004-0852
htget	0.93	Env. Var.	180	1.16/5.08	AEG's 0-day
expect	5.43	Env. Var.(HOME)	300	5.84/29.35	OSVDB-60979
expect	5.43	Env. Var.(DOTDIR)	300	6.10/29.28	AEG's 0-day
rsync	2.5.7	Env. Var.	201	2.17/9.92	CVE-2004-2093
acon	1.0.5	Env. Var.	1300	93.84/162.70	CVE-2008-1994
gif2png	2.5.3	Arguments	1080	65.24/154.67	CVE-2009-5018
hsolink	1.0.118	Arguments	1050	56.44/103.91	CVE-2010-2930
exim	4.41	Arguments	304	3.21/122.26	EDB-ID#796
aspell	0.50.5	Stdin	300	3.61/14.52	CVE-2004-0548
xserver	0.1a	Socket	104	1.16/14.35	CVE-2007-3957
xmail	1.21	Stdin	307	20.23/371.65	CVE-2005-2943

program can generate a jump-to-register exploit to bypass the ASLR. Only 0.06 seconds were spent on reasoning out the exploit while the total execution time was 3.16 seconds.

D. Real-World Programs Without Selective Symbolic Input

In the third part of the experiments, we generate exploits for real-world programs. Because real-world programs are larger and more complex than the sample code, this experiment demonstrates that our method is effective and practical in real-world applications.

We choose several programs from benchmarks of AEG, and three new vulnerable programs released in recent years, to perform this experiment. The 16 real-world programs are evaluated by an end-to-end approach, and the vulnerabilities of these programs are all stack buffer overflow. Concolic-mode simulation is used to perform exploit generation on all programs, and code selection intercept functions associating with file-related operations or pure error feedback, such as fopen() and perror(), to speed up the process. Table V shows the results of 16 real-word programs.

According to the results, the time was proportional to the length of the program input because the more symbolic data that exist, the more code may perform on symbolic execution.

TABLE VI
THE COMPARISONS OF DIFFERENT OPTIMIZATIONS OF CRAX

Program	Input Length	CRAX(T_1) Raw	CRAX(T_2) Fast Concolic	CRAX (T_3) Selective Input	Speed Up T_1/T_2	T_2/T_3	T_1/T_3
aeon	550	298.1	32.0	2.6	9.32	12.3	114.7
iwconfig	85	4.2	3.6	0.7	1.2	5.1	6.0
glftpd	300	50.1	8.0	0.5	6.3	16	100.1
ncompress	1050	2000.4	99.4	0.7	20.1	142	2857.7
htget(arg)	276	146.7	35.5	2.9	4.1	12.2	50.6
htget(env.var)	180	192.1	5.1	1.17	37.7	4.3	164.2
expect(HOME)	300	172.5	29.4	2.7	5.9	10.8	63.9
expect(DOTDIR)	300	169.3	29.3	3.56	5.8	8.2	47.6
rsync	201	210.53	9.9	2.7	21.2	3.6	77.9
acon	1300	3782.5	32.0	2.7	118.2	11.8	1400.9
gif2png	1080	12254.9	154.7	1.69	79.2	91.5	7251.4
hsolink	1050	2422.1	103.9	2.4	23.3	42.9	1009.2
exim	304	336.3	122.3	4.3	2.7	28.4	78.2
aspell	300	50.3	14.5	1.7	3.5	8.5	29.6
xserver	104	69.2	14.4	2.5	4.8	41.6	27.7
xmail	307	621.8	371.7	171.0	1.7	2.1	3.6

In addition, the longer the symbolic data will bring huge, complex constraints, and SMT solvers must spend a lot of time on constraint solving.

To reduce the overhead of the SMT solvers and speed up the process, code selection was used to concretize arguments of irrelevant functions. In this experiment, *aeon*, *htget*, and *acon* intercepted fopen(); *ncompress* intercepted __lxstat() and perror(); *gif2png* intercepted fopen() and perror(); *expect* intercepted open(); *rsync* intercepted vsnprintf(); *hsolink* intercepted system(). Those functions related with file operations often make constraint solvers stick, and the perror() function just print error messages without return values or influencing exploit generation, so we filtered these functions to speed up the process.

In this experiment, we performed exploit generation on real-world programs, and produced exploits for those applications successfully. The results show that the worst total time was about six minutes to generate an exploit for real-world programs, and the quality of exploits was good because they contained the longest NOP sled to increase the chances of successful attacks.

E. Constraint Optimization and Large Applications

In the ordinary concolic execution, the path constraints and the input constraint, along with the exploit constraints, are combined to be solved. We have tried to separate the constraint resolving processes, with two exclusive conditions: 1) path constraints and branch constraint, and 2) input constraint and branch constraint. The symbolic model construction process has minimal uses of the constraint solver, and in most of the cases, concrete value substitutions are used. We called this process the fast concolic process. After this process, along with selective symbolic input, the performance speed increase can achieve a multiple as high as 7251, as shown in Table VI. In Table VI, CRAX (raw) is the initial attempt without constraint reductions (indicated by T_1). CRAX(fast concolic, denoted as T_2), is with a

TABLE VII
THE RESULTS OF EXPLOIT GENERATION FOR LARGE PROGRAMS WITHOUT SELECTIVE INPUT

Program	Version	Executed Symbolic LOC	Constraint Size (bytes)	Path Exploring Time	Exploit Gen. Time
Unrar	2.90 beta 2 (linux 2.6.26)	1177301	2.91M	1388.45	2569.83
Mplayer	SVN-r33064 (Windows-XP)	1146887	3.89M	1713.76	2939.43
Foxit pdf reader	3.0 BUild 1301 (Windows-XP)	1825260	3.91M	5211.13	10094.17

single path concolic evaluation. CRAX(selective input, T_3) tries to filter out insignificant input as symbolic input.

The best speed increase between the initial CRAX and optimized CRAX is on gif2png. $T_1/T_3 = 7251$, where the fast concolic speed increase is 79, and the selective input speed increase is 92.

In Table VII, for the case of unrar, the original time is 3958 seconds, reduced to 13.5 seconds, with 293 times faster. The selective input method cannot reduce the symbolic input space of the *foxit* reader in the windows platform. The exploit generation process for foxit pdf reader still takes 4 hours, with the 1.8 M symbolic LOC executed, and 3.9 M constraints. It reveals that the automated process is made to symbolically resolve a large number of symbols, emulating the manual exploit writing process.

In Table VIII, we have summarized the exploit generation time for Linux and Windows platforms, and for different vulnerability types. All are optimized with the fast concolic and selective input methods.

F. Discussions

1) Comparisons of AEG Features: With the complete symbolic models supported by S^2E, CRAX uses less than 100 LOC to implement symbolic environment. And the concrete address indexed symbolic memory and selective symbolic execution are also built-in features of S^2E. We reduce the cost of failure model reconstructions by revising the KLEE symbolic executor, implementing S^2E plugins for selective code, path, and input, using about 6,000 LOC. In addition to simpler modeling, we do not make any assumptions on the failure situations. Our automatic exploit generation system supports various types of software crashes, including those introduced by tainted continuations, forged format strings, heap overflows by integer signedness, and uninitialized variable uses. The method is more general, without limiting the types of software vulnerability, and the recognition capability is delegated to the power of constraint solvers. The system we have developed is faster, even though the CRAX process is conducted in the whole system emulation level, compared with other methods in the process level. Our method can be at best 50 times faster than the existing systems by the selective input process (similar to the hot bytes finding process by *taintscope* and unique pattern generation by *metasploit*). The detailed comparisons are listed in Table IX.

2) Impacts of AEG Development:

- Exploit generation will become a powerful bug diagnosis technique. Viewing software crash as a tainted continuation, we detect exploits by producing symbolic continuations. That is, bug diagnosis is conducted by generating arbitrary process continuation to better understand the crash behaviors of various programs. Therefore, users can control the crash at will, manipulate the crash to the extent of their need (e.g., able to continue execution as usual), and dynamically patch the vulnerable running service, even if the system state has been corrupted.
- Zero-day exploit generation will become an automated process for average users, similar to Metasploit [22] acting as a shell code framework for common use. Stuxnet [27] like projects for physical infrastructure attacks can be developed easier than before.
- The link between software bugs and vulnerabilities can be bridged. Because 8lgm and rootshell announced the vulnerabilities in the form of real exploits, vulnerability is thought of as a security research topic, but few recognize that the vulnerability is closely related to software quality problems. CRAX urges an emphasis of quality assurance of released software, instead of releasing vulnerable software just for time-to-market consideration. Once crashes occur in uses of common users, an exploit is produced.
- Using a software bug as a backdoor becomes feasible. A sophisticated backdoor in a software program can be embedded as a symbolic continuation with a payload. We don't need to write explicit backdoors like the trapdoored login [28]. We can embed a bug for an implicit backdoor.
- Software security can be measured in terms of the software reliability and exploitability of the failure. Software reliability can be modeled as the mean time between failures. Exploitability can be expressed as the difficulty to build the failure symbolic model and exploit the failures, with a certain strength of mitigation strategies.

VII. RELATED WORK

APEG (Automatic Patch-based Exploit Generation) [29] compares the differences of a program between its buggy version and a patched version, and generates the exploits to fail the added check in the patched program. This work needs a patched version of the program, and is feasible only when the

TABLE VIII
THE RESULTS OF EXPLOIT GENERATION FOR REAL-WORLD PROGRAMS WITH DIFFERENT VULNERABILITY TYPES

Program	OS	Vulnerability Type	Input Source	Input Length	Exploit Gen. Time(sec.)	Source LOC	Advisory ID.
aeon	Linux	Stack Overflow	Env. Var.	550	2.6	797	CVE-2005-1019
iwconfig	Linux	Stack Overflow	Arguments	85	0.7	7,730	BID-8901
glftpd	Linux	Stack Overflow	Arguments	300	0.5	5,858	OSVDB-16373
ncompress	Linux	Stack Overflow	Arguments	1050	0.7	1,914	CVE-2001-1413
htget	Linux	Stack Overflow	Arguments	276	2.9	2,233	CVE-2004-0852
htget	Linux	Stack Overflow	Env. Var.	180	1.17	2,233	AEG's 0-day
expect	Linux	Stack Overflow	Env. Var.(HOME)	300	2.7	23,767	OSVDB-60979
expect	Linux	Stack Overflow	Env. Var.(DOTDIR)	300	3.56	23,767	AEG's 0-day
rsync	Linux	Stack Overflow	Env. Var.	201	2.7	25,094	CVE-2004-2093
acon	Linux	Stack Overflow	Env. Var.	1300	2.7	2,555	CVE-2008-1994
gif2png	Linux	Stack Overflow	Arguments	1080	1.69	1,834	CVE-2009-5018
hsolink	Linux	Stack Overflow	Arguments	1050	2.4	207	CVE-2010-2930
exim	Linux	Stack Overflow	Arguments	304	4.3	102,979	EDB-ID#796
aspell	Linux	Stack Overflow	Stdin	300	1.7	118	CVE-2004-0548
xserver	Linux	Stack Overflow	Socket	104	2.5	251	CVE-2007-3957
xmail	Linux	Stack Overflow	Stdin	307	171.0	50,366	CVE-2005-2943
a2ps	Linux	Stack Overflow	Stdin	550	11.5	16,179	EDB-ID-816
freeradius	Linux	Stack Overflow	Stdin	9000	18.2	33,457	MAYHEM's zero day
htpasswd	Linux	Stack Overflow	Arguments	400	0.4	3,810	OSVDB-ID-10068
mbse-bbs	Linux	Stack Overflow	Env. Var.	4200	26.9	139,010	CVE-2007-0368
psutils	Linux	Stack Overflow	Arguments	300	25.4	1,968	EDB-ID-890
squirrel mail	Linux	Stack Overflow	Arguments	150	0.9	264	CVE-2004-0524
socat	Linux	Format String	Arguments	600	114.3	16576	CVE-2005-2943
tipxd	Linux	Format String	Arguments	250	23.9	2,257	OSVDB-ID-12346
orzhttpd	Linux	Format String	Network	400	22.2	2,980	OSVDB-ID-60944
Unrar	Linux	Stack Overflow	Arguments	550	13.5	17,005	EDB-ID-17611
MPlayer	Linux	Stack Overflow	File	298	3.6	625,204	CVE-2008-0630
NullHTTPd	Linux	Heap Overflow	Socket	417	58.9	1,978	CVE-2002-1496
Coolplayer	Windows	Stack Overflow	File	210	25.2		CVE-2008-3408
Distiny	Windows	Stack Overflow	File	2100	47.1		OSVDB-ID-53249
Galan	Windows	Stack Overflow	File	1500	15.9		OSVDB-ID-60897
GSPlayer	Windows	Stack Overflow	File	400	29.2		OSVDB-ID-69006
Mplayer	Windows	Stack Overflow	File	5568	167		EDB-ID-17013
Foxit	Windows	Stack Overflow	File	10503	15,305.3		CVE-2005-2943
netapi32.dll	Windows	Uninitialized Uses	Arguments	1060	170.0		CVE-2006-3439
Microsoft word	Windows	Stack Overflow	File	51643	119.1		CVE-2010-3333
Microsoft word	Windows	Stack Overflow	File	7307	216.2		CVE-2012-0158

patch is to add input sanitization logic. In addition, most of the exploits generated by APEG are DoS (Denial-Of-Service) attacks, which just crash a program, without executing shellcode or malicious tasks.

AEG (Automatic Exploit Generation) [2] generates exploits in two stages: finding bugs on symbolic execution, and then collecting run-time information on concrete execution. AEG only deals with stack buffer overflow and format string vulnerability because it has to add individual safety check constraints to detect each bug. Furthermore, AEG implements an end-to-end approach for exploit generation, including symbolic files, symbolic sockets, etc., and uses return oriented programming to bypass both W $\oplus$ X and ASLR[4]. MAYHEM [16] is the first binary AEG, implementing many binary AEG features in PIN. However, many are the built-in features in S^2E, including selective path, symbolic environment, and concrete address mapped symbolic memory. In our CRAX implementation, we just reuse these features.

Heelan *et al.* [3] uses binary instrumentation to perform taint propagation, and collect runtime information. Their work generates exploits by checking whether the EIP register is corrupted by a tainted value, and also handles pointer corruption that corrupts the EIP register indirectly. Similar to our work, a crashing input is needed for taint analysis.

In addition, some systems do not generate exploits explicitly, but aim to report a bug which is probably exploitable. For ex-

TABLE IX
COMPARISON OF AEG FEATURES

System/ Features	Heelan's Sep 2009	CMU's AEG Feb 2011	MAYHEM May 2012	CRAX June 2012
AEG	Yes	Yes	Yes	Yes
Vulnerabilities	Stack overflow	Stack overflow/format string	Stack overflow/format string	Stack/heap overflow,format string, uninitialized uses
End-to-end	No	Yes	Yes	Yes
Source/Binary	Source	Source	Binary	Binary
Instrument	PIN	KLEE	PIN	S^2E(QEMU+LLVM+KLEE)
Symbolic Environment	Partial	Partial	Partial (30 system calls)	6 S^2E models, all environments
Symbolic Memory	-	Abstract Address	Concrete Address	Concrete Address
Selective Symbolic Execution	-	-	Selective Path	Selective Code/Path/Input
Performance	-	Fast	Slow	Faster (larger and much faster, 10 times faster)
Scale	XBMC	Xmail	Dizzy	office word/mplayer/foxit pdf reader
Platforms	linux	linux	linux/windows	linux/windows/web
Applicability	process	process	process	process/system/kernel

ample, !exploitable [8], and some projects [6] of BitBlaze, analyze a crash, and determine whether it is exploitable.

VIII. CONCLUSIONS

In this paper, we implement an automated exploit generation framework, called CRAX, which is built on S^2E, a new platform for symbolic execution. To generate control flow hijacking attacks, we focus on detection of the symbolic EIP, other continuation based registers, and pointers; and we propose a systematic method for searching maximum contiguous symbolic memory for payload injection. Detection of symbolic registers is a comprehensive, easier way to deal with all kinds of control flow hijacking vulnerabilities.

We implement concolic-mode simulation to perform concolic testing on symbolic execution so that switching between symbolic execution and concolic testing is easy without modifying the memory model. In addition, code selection helps S^2E to filter irrelevant functions, and thus enables symbolic execution to explore interested code more effectively, and accelerate the process of exploit generation. We also use selective symbolic input to reduce the symbolic variable size by identifying the significant inputs (called hot bytes) that will influence the continuations. By only marking these bytes, the whole exploit generation time can be reduced significantly. We also propose the concept of pseudo symbolic variables and lazy evaluation to handle symbolic pointers. We can automatically exploit the unlink() macro of glibc.

To evaluate CRAX, we conducted experiments on a variety of vulnerable sample code to demonstrate that it can tackle different kinds of control flow hijacking vulnerabilities. We also experimented on real-world large programs, and generated *return-to-libc* and *jump-to-register* exploits to bypass mitigations of ALSR or $W \oplus X$ in real-world software. The successes on *mplayer*, and *foxit* pdf reader show that CRAX is a feasible, powerful exploit generation tool for real environments.

A. Future Work

CRAX can be extended for larger programs, including Open-office, Microsoft Office, and popular web browsers. Driver mode applications like anti-virus software in windows and kernel modules like VM hypervisors are also our future targets for potential exploitations. These targets are not possible for process level AEG like MAYHEM, but feasible for S^2E-based AEG like CRAX. Because CRAX can be applied to the whole system, we have preliminarily tested web systems: apache with the php module, and mysql server. Currently, we have developed a web exploit generation tool, called CRAXweb [30], able to automatically generation SQL injection and cross-site scripting attacks from php and python web applications. More web vulnerability types and web platforms, including asp, jsp, and ruby will be supported.

REFERENCES

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ, USA: Pearson Education, 2007.
[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proc. Netw. Distributed Syst. Security Symp. (NDSS'11)*, Feb. 2011, pp. 59–66.
[3] S. Heelan and D. Kroening, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," MSc Computer Science, Univ. Oxford, London, U.K., 2009.
[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. 20th USENIX Security Symp. (USENIX'11)*, Aug. 2011.
[5] M. Martin and M. Lam, U. Association, Ed., "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *Proc. 17th Conf. Security Symp.*, 2008, pp. 31–43.
[6] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Crash analysis using BitBlaze," in *Proc. Black Hat*, Jul. 2010.
[7] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information Systems Security*. Berlin, Germany: Springer-Verlag, 2008, pp. 1–25.
[8] *Exploitable Crash Analyzer – MSEC Debugger Extensions*, [Online]. Available: http://msecdbg.codeplex.com/
[9] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 430–447, 2011.
[10] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. 31st IEEE Symp. Security Privacy (S&P 2010)*, 2010, pp. 317–331.
[11] P. Team, Pax Address Space Layout Randomization (ASLR) 2003 [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt
[12] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 552–561.
[13] V. Ganesh and D. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. 19th Int. Conf. Comput. Aided Verification (CAV'07)*, 2007, pp. 519–531.

[14] J. C. Reynolds, "The discoveries of continuations," *Lisp and Symbolic Comput.*, vol. 6, no. 3-4, pp. 233–247, 1993.
[15] C. Strachey and C. P. Wadsworth, "Continuations: A mathematical semantics for handling full jumps," *Higher-Order Symbolic Comput.*, vol. 13, no. 1, pp. 135–152, 2000.
[16] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing MAYHEM on binary code," in *IEEE Symp. Security Privacy*, 2012, pp. 380–394.
[17] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Arch. Support Programm. Lang. Operating Syst. (ASPLOS'11)*, Mar. 2011, pp. 265–278.
[18] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Symp. Operating Syst. Design Implement. (OSDI'08)*, Dec. 2008, pp. 209–224.
[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. FREENIX Track: 2005 USENIX Annu. Technical Conf.*, Apr. 2005, pp. 41–46.
[20] D. A. Molnar and D. Wagner, Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors Univ. California, EECS Department, Berkeley, CA, USA, 2007, Tech. Rep. UCB/EECS-2007-23.
[21] S. Hocevar, ZZUF – Multi-Purpose Fuzzer [Online]. Available: http://caca.zoy.org/wiki/zzuf
[22] H. D. Moore, The Metasploit Project [Online]. Available: http://www.metosploit.com
[23] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 497–512.
[24] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. 10th Eur. Software Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Foundat. Software Engin. (ESEC/SIGSOFT FSE'05)*, Sep. 2005, pp. 263–272.
[25] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. 2005 ACM SIGPLAN Conf. Programm. Lang. Design Implem.*, 2005, pp. 213–223.
[26] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Software Engin.*, 2008, pp. 443–446.
[27] T. Chen, "Stuxnet, the real start of cyber warfare? [editor's note]," *IEEE Netw.*, vol. 24, no. 6, pp. 2–3, 2010.
[28] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984.
[29] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. IEEE Symp. Security Privacy*, May 2008, pp. 143–157.
[30] S. K. Huang, H. L. Lu, W. M. Leong, and H. Liu, "Craxweb: Automatic web application testing and attack generation," in *Proc. IEEE Int. Conf. Software Security Rel. (SERE)*, 2013, pp. 208–217.

Shih-Kun Huang is a faculty member in the Information Technology Service Center, and jointly with the Department of Computer Science at the National Chiao Tung University in Hsinchu, Taiwan. His research interests are in open source software engineering, object-oriented technology, and software quality. He received his B.S., M.S., and Ph.D. degrees in Computer Science and Information Engineering from the National Chiao Tung University in 1989, 1991, and 1996 respectively.

Min-Hsiang Huang received the B.S., and M.S. degrees in Computer Science and Engineering from the National Chiao Tung University, Taiwan in 2008, and 2010 respectively. He is currently pursuing the Ph.D. degree at the Institute of Computer Science and Engineering of National Chiao Tung University. His research interest is in compilation techniques on hetrogeneous platforms.

Po-Yen Huang received the B.S. degree in Computer Science and Information Engineering from Providence University, Taiwan in 2008, and the M.S. degree from the National Chiao Tung University, Taiwan in 2011. His research interests include software vulnerability detection, and exploit generation. He is currently a Firmware R&D Engineer in the Firmware R&D Center of ASUSTeK Computer Inc.

Han-Lin Lu received the B.S. degree in the Department of Transportation Technology and Management, and M.S. degree in Computer Science and Engineering from the National Chiao Tung University, Taiwan in 2010, and 2012 respectively. He is currently pursuing the Ph.D. degree at the Institute of Science in Computer Science and Engineering of National Chiao Tung University. His research interests include software quality, network security, and software security.

Chung-Wei Lai received the B.S. degree from the Department of Information Technology and Management from Shih Chien University, Taiwan in 2010, and M.S. degree from the National Chiao Tung University, Taiwan in 2012. He is now a Communication R&D engineer at ASUS. His research interest is in software vulnerability detection.