2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Hercules: Reproducing Crashes in Real-World Application Binaries

Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
Email: thuanpv@comp.nus.edu.sg, noodiew1@yahoo.com.sg, rubinov@comp.nus.edu.sg, abhik@comp.nus.edu.sg

***Abstract*—Binary analysis is a well-investigated area in software engineering and security. Given real-world program binaries, generating test inputs which cause the binaries to crash is crucial. Generation of crashing inputs has many applications including off-line analysis of software prior to deployment, or online analysis of software patches as they are inserted. In this work, we present a method for generating inputs which reach a given "potentially crashing" location. Such potentially crashing locations can be found by a separate static analysis (or by gleaning crash reports submitted by internal / external users) and serve as the input to our method. The test input generated by our method serves as a witness of the crash. Our method is particularly suited for binaries of programs which take in complex structured inputs. Experiments on real-life applications such as the Adobe Reader and the Windows Media Player demonstrate that our Hercules tool built on selective symbolic execution engine S2E can generate crashing inputs within few hours, where symbolic approaches (as embodied by S2E) or blackbox fuzzing approaches (as embodied by the commercial tool PeachFuzzer) failed.**

I. Introduction

Complex software systems are released and deployed with faults. Some faults trigger application crashes that elevate system security risks and are difficult to trace, analyze and reproduce. The problem of finding crashing paths has been addressed by previous research, however, few techniques cope with large real-world binaries. Real-world binaries present challenges for program analysis techniques due to their size, complexity, and multitude and depths of execution paths. In addition, the information about structure of programs in a stripped binary is incomplete when collected statically, while recovering such information dynamically is often infeasible.

Reproducing crashes in multi-module systems requires targeted exploration. The search space of potential crashing paths is too large to be exhaustively checked path by path (for instance, using symbolic execution); the complexity of program inputs is too high for exhaustive set of inputs to be generated combinatorially or randomly (for instance, using fuzzing). The space of program paths is intractable for modern analysis techniques – a novel targeted exploration is needed.

Given a real-world program binary with a crash report, our approach *Hercules* tackles the problem of finding program paths and corresponding program inputs that cause a crash in a given program location. The **core idea** behind our approach is to systematically detect, bound and explore a subset of program paths necessary and sufficient for reaching and triggering a given program crash. The approach builds upon concolic exploration and propagates a necessary, but minimal subset of input data in symbolic form, while keeping the remaining input data concrete. The exploration uses targeted search strategy that helps to explore as few as possible paths, while resolving enough information about program structure for finding the crashing path.

Our approach works in **three main steps** (Figure 1). Each step progressively more precisely establishes program and input structures that are relevant to reproducing the crash. The approach starts with a preprocessing step for initial reconstruction of a program structure and selection of input files (*Step 1*) followed by two passes of concolic exploration. Application of two passes of concolic exploration is a distinct feature of the approach. Each pass serves different purpose: the first pass (*Step 2*) establishes a relationship between program input and relevant program structures, and provides an input to the second pass (*Step 3)* – a focussed fine-granularity search for a crashing path. Our search strategy infers the reasons for infeasibility of specific non-crashing paths, thus helping us to avoid exploring large numbers of paths that are non-crashing for the same reason, and to direct the search towards the paths that will crash the system.

We call our tool and method as *Hercules*, largely because of the Herculean task (of finding crashing inputs) it accomplishes in a reasonable time-frame. This is because of smart search heuristics and structuring of the search into phases. Our approach builds upon selective symbolic execution technique *S2E* [1], extends it, and makes a number of technical contributions, namely -

- *Targeted search strategy* implements our targeted search algorithm that detects reasons for infeasibility of non-crashing paths and directs concolic execution.
- *Approximation of string functions* scales concolic execution by bounding exploration of string manipulation functions that generally cause path explosion.
- *Analysis of loop-controlled crash instructions* enables automatic synthesis of loop-dependent crash conditions.
- *Dynamic module selection* adds flexibility to the *S2E* technique – in the process of selective concolic execution our technique allows dynamic selection of program structures for concolic execution (state forking).
- *Dynamic CFG refinement*. In addition to the standard *S2E* functionality our technique builds and dynamically refines program control flow graph (CFG) and uses it for reachability analysis to inform concolic exploration.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.99

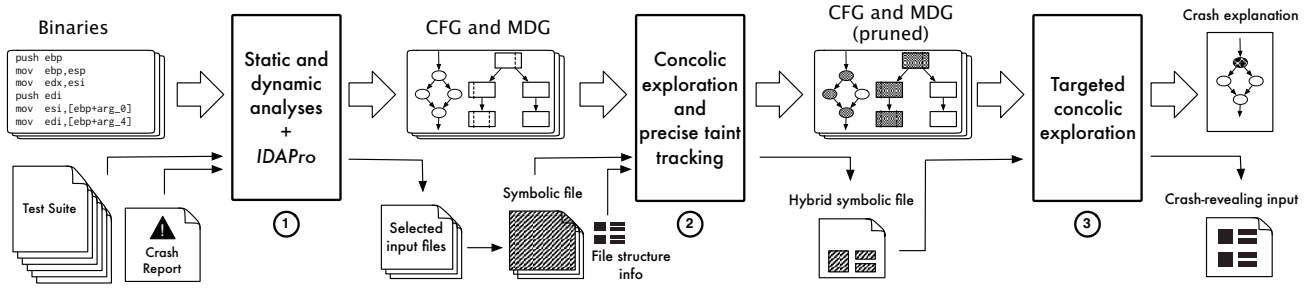

Fig. 1. An overview of our approach and *Hercules* tool

Assumptions: The few significant assumptions we make concern availability of a test suite (a set of non-crashing benign input files) `TestSuite` and indication of a crash location `CL` in a crashing module `CrashingModule`, where execution of at least one test case in `TestSuite` reaches `CrashingModule`. We optionally assume availability of a list of modules invoked during crashing execution `ModuleList`, call stack and values of the program registers at the moment of crash that form a crash condition `CC`. If available, the knowledge about input file structure and layout may aid seed file selection and generation of hybrid symbolic inputs. The required information is external to the approach and can be often produced by a separate static/dynamic analysis.

II. Overview

We illustrate the pertinent aspects of the approach using data from a vulnerability `CVE-2010-0718` in Windows Media Player – a buffer-overflow that triggers a system crash in a divide-by-zero exception. Figure 2 shows fragments of information used by our approach in the search for crashing input. According to the crash report, the list of modules involved in crashing behavior contains *quartz.dll*, *wmp.dll* and a main module *wmplayer.exe* (out of total 84 modules loaded by the program).[1] The module *quartz.dll* crashes at the program location `0x74902224`, instruction '`div ecx`'. A set of benign inputs does not reach this location.

In *Step 1* of the approach we reconstruct the structure of a system with static analysis and dynamically, by exploring the system with benign input files. The result of this step is an incomplete program structure incorporating module dependence and control flow information. *Step 1* also selects benign inputs that trigger execution in the modules involved in the crash. For `CVE-2010-0718`, *Step 1* identifies a benign input that reaches the crashing module at an entry point (internal function `0x74834010`), but does not reach the crash location. A fragment of the benign file is shown in Figure 2.

In *Step 2*, we use concolic exploration as an apparatus for precise taint tracking and identifying input fragments relevant to reaching the crashing module. From these data we generate hybrid symbolic inputs that maintain correct input

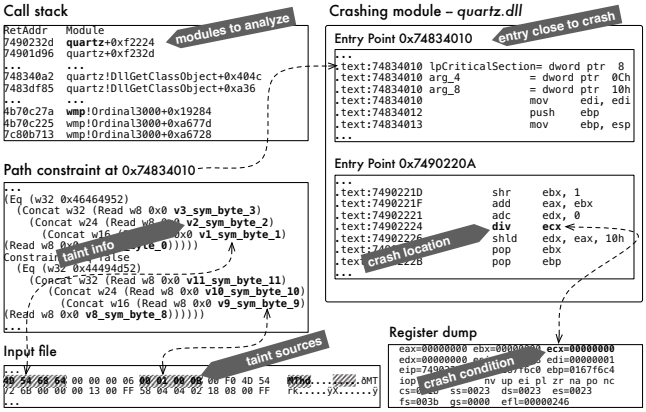

Fig. 2. Crash analysis information for `CVE-2010-0718`

file structure.[2] For `CVE-2010-0718`, *Step 2* collects a path constraint with a symbolic version of the benign input. The path constraint indicates symbolic bytes from the input file that are propagated to the module entry point `0x74834010` (arrows between the path constraint and the input file in Figure 2). These input portions are relevant to reaching crashing module and we mark them symbolic in a hybrid symbolic file.

Generation of hybrid symbolic inputs addresses two main issues in symbolic execution for real-world program binaries. First, hybrid symbolic inputs prompt less constraint solving in concolic exploration and result in smaller symbolic formulae. Second, exploration with structurally correct hybrid inputs has higher chances of bypassing the parser component that incorporates multitude of conditions that cause state explosion in symbolic execution and prevent it from reaching deep program paths.

In *Step 3*, we apply a targeted search strategy (second pass of concolic execution) to explore the system in a directed fashion systematically eliminating groups of paths from analysis and generate crashing input. The strategy stems from the observation that groups of non-crashing paths often have the same *cause* for which they *do not* crash the system. The **main intuition** behind our strategy is that we can detect a reason of infeasibility of a certain path and eliminate from consideration in concolic execution groups of paths that do not crash for

[1] We refer to a *module* to denote an executable file (main module) and any library it loads, while for the *entry points* of a module we consider both exported and internal functions.

[2] We refer to *hybrid symbolic* inputs as files containing fragments of symbolic and concrete data, in contrast with *fully symbolic* files that contain only symbolic data.

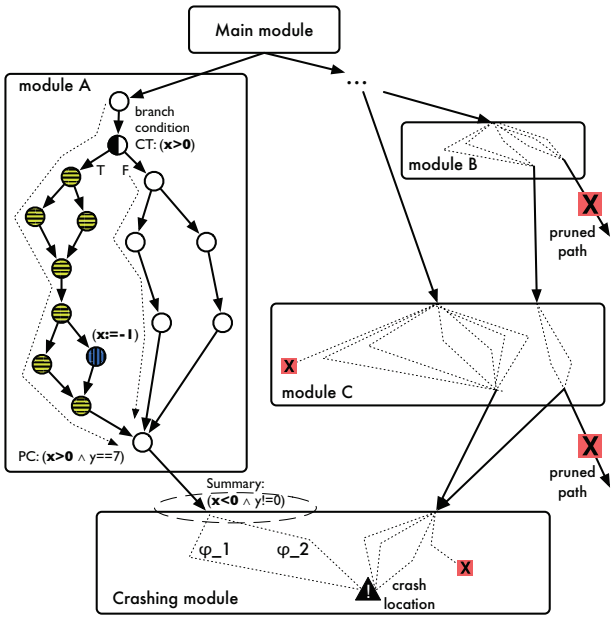

Fig. 3. Schematic module dependence graph with paths to crashing module highlighted

the same reason. A contrived example of a shared cause for non-crashing paths is shown in Figure 3. Paths through nodes highlighted in yellow (horizontal bars) cannot crash the system because they are guarded by the condition `(x>0)` that does not satisfy the crashing condition `(x<0 ∧ y!=0)`.

To detect the reason of infeasibility of non-crashing paths, we conjoin the path constraint ϕ for a path that reaches crashing module with the symbolic summary Σ of a crashing module with respect to crash location. Intuitively, terms in path constraint ϕ that contradict terms in symbolic summary Σ are the reasons of infeasibility of a complete path from program entry point to the crash location (term `(x>0)` in path constraint `PC` in Figure 3).

Practically, the conjoining of ϕ and Σ amounts to two steps. First, to check the satisfiability of formula $\phi \wedge \Sigma$. And second, if the formula is not satisfiable (the path does not crash the system), to extract a minimal `unsat_core` that contains contradicting terms T. The contradicting terms correspond to the causes of the infeasibility of a given non-crashing path.

Our search algorithm keeps track of each term in path constraint formula and corresponding program location the term is being introduced. Consequently, a contradicting term indicates a point on a program execution path to which our search algorithm proceeds to pursue alternative paths and avoids executing paths that do not crash for the same identified reason.

Crash reports often contain the values of program registers at the moment of crash (register dump in Figure 2). A constraint on these values is a crash condition. In case crash condition is available, we can detect the reasons of infeasibility of a path with respect to specific crash condition CC in the same way as described above, by extracting terms in `unsat_core` from an unsatisfiable formula $\phi \wedge \Sigma \wedge CC$.

Depending on a type of a crash, crash conditions are easier or more difficult to extract. For instance, crash due to division by zero could appear in crash report as instruction `'div ecx'`, where the value of `ecx=00000000` as shown in Figure 2. Consequently crash condition is `(ecx==0)`. Some crash conditions can be less evident and require additional effort for being captured as we discuss in Section V-A.

III. Preprocessing and Hybrid Symbolic Inputs

The first two steps of our approach prepare information for the third step – a targeted search for a crashing path (Section IV). In particular, the first step resolves incomplete information about program binaries, while the second step generates a structurally correct hybrid symbolic input (Figure 1).

Step 1: Recovering program structure and selecting seed files

We statically analyze the system with *IDA Pro* toolset[3] (https://www.hex-rays.com/idapro/) and use analysis results to obtain a program control flow graph (CFG) and module dependence graph (MDG) that we dynamically refine in the next steps of the approach. The main sources of incompleteness in program binaries are register indirect jumps and calls, and concealed library entry points (non-exported functions). We process the output of *IDA Pro* and statically resolve jump targets for `switch` statements, detect function boundaries and statically imported entry-exit points for the modules in a list of modules involved in a crash.

We execute the system with benign input files to augment the statically collected information with dynamically imported entry and exit points of the modules of the system, concrete targets for branches dependent on indirect register jumps, and resolved register indirect call targets. A resulting aggregated inter-procedural control flow graph connects different modules of the system along the discovered module entry-exit points.

We select seed files from a test suite according to their relevance to the crash and the modules involved in the crashing behavior. The main criteria for file selection are traces of system executions with test files and file structure information. File structure information indicates which objects in the file are required to exercise certain functionality of the system. We aggregate the traces of system execution with the preselected test files and obtain a histogram for selecting files that most extensively use modules from `ModuleList`. We use the selected seed files in the next steps of the approach as the most relevant to the crashing behavior.

Step 2: Generating hybrid symbolic inputs

We use concolic execution to detect fragments of inputs that are relevant to reaching the crashing module – input fragments that propagate data into the crashing module. Taint tracking using concolic execution precisely associates the fragments of program inputs and affected program locations. It is more accurate than "vanilla" taint analysis that traces program paths affected by program inputs, however does not establish which parts of the input are propagated to which program locations.

[3] *IDA* is a state-of-the-art multi-processor disassembler and debugger

In concolic execution we apply random exploration strategy – upon branching, the next path to explore is selected randomly, with an exception that paths generated by string functions are selected from groups of paths as detailed in Section V-B. We automatically generate fully symbolic versions of seed files identified in the previous step of the approach and we trace the propagation of symbolic data from these files during concolic execution; execution stops when it reaches `CrashingModule`.

Given a path that reaches crashing module, a path constraint contains symbolic input bytes (taint sources) relevant for reaching this module. Together with the knowledge of input file structure, this information serves to automatically generate hybrid concolic input file that maintains the original file layout.

We prevent random exploration from "drifting" outside modules in `ModuleList` and dynamically refine CFG of the system extending it with information from concolic exploration. Concolic exploration discovers new paths if it produces new concrete data to take these paths. In particular, if concolic executor reaches register-indirect jump instruction `'jmp [eax]'` with a *new* concrete value in `eax`, then it may explore a new path spanning from a new jump target. A maximum number of resolved indirect jumps and calls is thus proportional to the number of new paths we can explore with the concolic data. To prepare CFG for the targeted search, we prune it with respect to crash location in crashing module and with respect to exit points that connect modules in `ModuleList`. A schematic module dependence graph with pruned paths is shown in Figure 3, where pruned paths are marked with **X**.

IV. Targeted search

In the third step of the approach we apply targeted concolic execution to find crashing paths – program paths that crash the system in `CrashingModule`. The targeted exploration works on a pruned version of CFG and hybrid concolic inputs generated in the previous step of our approach. The three phases of the exploration are: (1) *replay*, (2) *summarization*, and a main phase – (3) *targeted search*. Figure 4 illustrates transitions between these phases schematically.

The targeted exploration starts by deterministically replaying one of the observed paths to the crashing module with hybrid concolic input (*replay*). Consecutive symbolic exploration symbolically summarizes the crashing module from module entry point to the crashing location using symbolic data propagated to the module from the program input (*summarization*). Finally, a *targeted search* phase selects and traverses alternative program paths in search for crashing paths.

Targeted search phase evaluates the feasibility of a given path with respect to the crashing module summary and detects the reasons of infeasibility of the non-crashing path as terms in the `unsat_core` of the conjunction of the path constraint and module summary. The algorithm uses the program states that introduced the infeasibility reasons as anchors to select alternative states to which to proceed. As a result, the search is directed away from groups of infeasible paths. Search proceeds

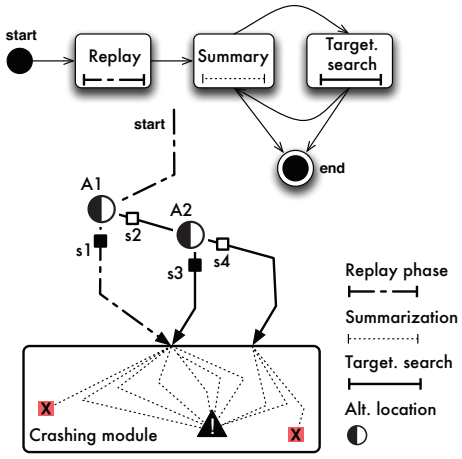

Fig. 4. Phases of targeted exploration

until it finds a feasible crashing path or terminates after a user-specified timeout or upon exhausting the memory.

Algorithm 1 outlines the key elements of each of the phases. The search algorithm is general and can be implemented on top of any dynamic symbolic executor. We illustrate the algorithm for a generic language with instructions identified by their location l. For simplicity we distinguish two types of instructions: (1) branches identified by $branch(l)$ predicate with branch condition $cond(l)$, and (2) non-conditional instructions. The target location of a branch instruction is identified by $target(l)$, while for all instructions the next location is $next(l)$.

Program state s is represented by a triple (l, ϕ, m), where l is a program location, ϕ is a path constraint, and m is a symbolic store. Symbolic store maps program variables to concrete values or expressions over input variables. The initial program state is $(l_0, true, m)$, where l_0 is a program entry point, path constraint is set to $true$, and m is initialized with symbolic variables for each program input variable.

A. Replay

Targeted exploration replays the path to one of the crashing module entry points $e \in E$. Given a set of states S_e (list of states for reaching e from l_0), the replay is a concolic exploration where upon branching the states for execution are selected from S_e (line 8). During replay the searcher takes snapshots of the alternative states and stores them in a map μ with constraints introduced in the executed state $condition(s)$ (line 9). Figure 4 schematically shows state `s1` and its alternative state `s2` that the algorithm stores in the map μ. Replay terminates after traversing all the states in S_e in a state reaching entry point e of the crashing module.

Note that in our implementation of the algorithm, S_e list is lightweight. It does not store the complete state representation as used by *S2E*, but only the forking program locations.

B. Summarizing crashing module symbolically

Upon reaching entry point of the crashing module, the algorithm commences symbolic summarization (line 11, lines 28–43). The summary is an aggregate of path constraints for

Algorithm 1 Targeted search

Input: l_0 – initial location; χ – crash location; E – list of module entry points; S_e – list of states for reaching $e \in E$ from l_0;

```
    // REPLAY PHASE:
 1: s ← (l_0, true, m)                                   ▷ Initialize current state
 2: while S_e ≠ ∅ do
 3:     if ¬branch(l) then s ← (next(l), φ, m⟨v, e⟩)
 4:     if branch(l) then
 5:         if (SAT(cond(l) ∧ φ) ∧ SAT(¬cond(l) ∧ φ)) then
 6:             s_1 ← (next(l), cond(l) ∧ φ, m)
 7:             s_2 ← (target(l), ¬cond(l) ∧ φ, m)
 8:         s ← {s_1, s_2} ∩ S_e                        ▷ Pick next state from S_e
 9:         μ ← μ⟨condition(s), ({s_1, s_2} \ s)⟩       ▷ Snapshot
10:     S_e ← S_e \ s
11: Σ ← SymbSummary(s)          ▷ Location of the last state in S_e is e
    // MAIN PHASE:
12: X ← ∅
13: while ¬SAT(φ ∧ Σ) do
14:     τ ← UNSAT_CORE(φ ∧ Σ)
15:     t ← pickTerm(τ, μ)      ▷ Pick contradicting term using strategy
16:     X ← X ∪ {t}
17:     s ← μ[t]                                        ▷ Select alternative state
18:     while l ∉ E do                  ▷ Until reached any of entry points
19:         if ¬branch(l) then s ← (next(l), φ, m⟨v, e⟩)
20:         if branch(l) then
21:             if (SAT(cond(l) ∧ φ) ∧ SAT(¬cond(l) ∧ φ)) then
22:                 s_1 ← (next(l), cond(l) ∧ φ, m)
23:                 s_2 ← (target(l), ¬cond(l) ∧ φ, m)
24:             s ← pickNextState(s_1, s_2)
25:             μ ← μ⟨condition(s), ({s_1, s_2} \ s)⟩   ▷ Snapshot
26:     if l ∉ E_checked then Σ ← SymbSummary(s)
27: OUT ← (φ, X)        ▷ We can continue search by proceeding to the
                          remaining alternative states from line 15.
    // SUMMARIZATION
28: procedure SymbSummary(s)                  ▷ Explore paths from s to χ
29:     Require: location(s) ∈ E
30:     E_checked ← E_checked ∪ location(s)
31:     s ← (l, true, m)                                ▷ Reset path constraint
32:     W ← {s}                                          ▷ Initialize worklist
33:     while W ≠ ∅ ∨ timeout do
34:         if ¬branch(l) then W ← W ∪ (next(l), φ, m⟨v, e⟩)
35:         if branch(l) then
36:             if (SAT(cond(l) ∧ φ) ∧ SAT(¬cond(l) ∧ φ)) then
37:                 W ← W ∪ (next(l), cond(l) ∧ φ, m)
38:                 W ← W ∪ (target(l), ¬cond(l) ∧ φ, m)
39:         if l == χ then Σ ← Σ ∨ φ
40:         W ← W \ s
41:         s ← pickNextState(W)
42:     Σ ← Σ ∧ CC                          ▷ Add crash condition to the summary
43:     return Σ
```

each path reaching crash location from the module entry point using symbolic data that is propagated to the module entry point. Symbolic store m holds the propagated symbolic data as expression over symbolic program inputs. The summary is independent of the path constraint ϕ used for reaching the crashing module and the corresponding path constraint is reset to $true$ (line 31). A module summary Σ is a disjunction of path constraints φ_i for each path reaching crashing location χ from a given module entry point: $\bigvee_{i=1}^{n} \varphi_i$.

Summarization procedure uses the pruned CFG to inform selection of the next states in symbolic exploration. States extending outside CFG are not pursued as they do not reach crashing location. This is implemented in procedure $pickNextState()$ that uses CFG to select successors for branching instructions (line 41). This way the algorithm ensures selection of states for paths that reach crashing location.

The symbolic summary collected with our approach may be incomplete. Symbolic data in symbolic execution can be injected only from the input of the system – it is not be generated in the process of symbolic execution. Concolic exploration may not reach the module with symbolic data for all of its inputs, some of the inputs may be reached with concrete data resulting in an incomplete summary.

Symbolic data may not reach the module for a number of reasons. First, seed input files may be inadequate or deficient with respect to the functionality of a crashing module, input file may lack data structures that affect certain input of a crashing module. Second, an input of the module may be independent of the program input. And third, a symbolic input may be concretized during concolic execution and propagated to the module input as a concrete data.

Given a crash condition CC, a module summary Σ is a precondition with respect to reaching crashing location, where $\Sigma(CrashingModule, CC)$ is a logical formula over the module input which is true for all inputs that cause crashing module to reach a final state satisfying CC. Since CFG of crashing module is pruned, module final state is in the crashing location χ. The algorithm extends module summary Σ with crash condition CC in the last step of summarization (line 42). The summary Σ concisely captures a precondition for reaching crashing location.

C. Searching for a crashing path

Targeted search phase starts from the point when concolic executor have reached the crashing module in the *replay* phase and consequently collected symbolic summary Σ of the module in the *summarization* phase. *Targeted search* phase identifies program states that *do not* introduce infeasible constraints in the paths reaching crashing module and directs exploration through these states in the search for feasible crashing paths.

Provided that the initial path selected for reaching the crashing module in the replay phase does not crash, the conjunction of path constraint and symbolic summary $\phi \wedge \Sigma$ is unsatisfiable. To detect the reasons of unsatisfiability the algorithm queries SMT solver for minimal `unsat_core` that contains a list of contradicting terms from both path constraint ϕ and summary Σ. The algorithm extracts from `unsat_core` a list of terms τ (line 14). These terms correspond to the reasons for infeasibility that originate from the specific program states on the path reaching crashing module. In the schematic example in Figure 3 the cause for the path infeasibility is located by the contradicting term `(x>0)` from the path constraint.

To continue the search for a crashing path, the algorithm selects alternative program states that do not introduce the identified infeasibility reasons. The algorithm selects alternative states indicated by the list of contradicting terms τ using the map of constraints and alternative state snapshots μ captured during replay phase. In particular, a procedure $pickTerm(\tau, \mu)$ selects one term t from the list τ and this term is then used to query the map μ to select the alternative state (lines 15–17).

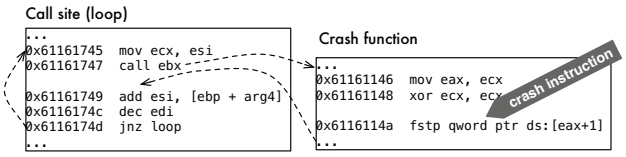

Fig. 5. Example of loop-dependent crash in Real Player

In $pickTerm(\tau, \mu)$ we select a term introduced in the top-most program location and a corresponding alternative state. Such term represents a general reason for infeasibility of multiple paths in a symbolic subtree and thus, when selected, can dramatically reduce the search space. However, some of the paths in that subtree may be feasible. For instance, in Figure 3, a path that passes through a blue node (vertical bars) in the CFG is feasible with respect to the crashing module summary.

Each iteration of the targeted exploration continues from the selected alternative state until it reaches entry point of the crashing module with a new path constraint ϕ. The search algorithm can reach crashing module through an entry point that it has not reached before (line 26). In this case the module summary is recomputed to consider new paths to the crashing location, if they are reachable from this entry point.

Algorithm 1 iterates until the formula $\phi \wedge \Sigma$ is satisfiable and hence the crashing path is found. The output OUT of the targeted search consists of a path constraint ϕ and a list of contradicting terms X used for navigating the search. The path constraint ϕ can be solved to generate a set of program inputs that exercise a particular crashing path. The list of selected contradicting terms X serves as an additional explanation for the crashing path highlighting the data and deviation points (`A1` and `A2` in Figure 4) that are crucial for pursuing it.

V. Tackling limitations of concolic execution

A. Synthesizing crash conditions for loop-controlled crashes

To reproduce a crash our approach reaches a crash instruction and, among other information, uses crash condition CC to direct the targeted search and, ultimately, synthesize crashing input. In practice, however, crash condition cannot be formulated symbolically in terms of symbolic input of the program if concolic executor reaches crashing instruction without symbolic data in the operands of the instruction. Previous research demonstrated that this situation can be alleviated for loop-dependent variables [2].

Hercules solves this problem by inferring a function over dependent variables (operands of crash instruction) on a number of loop iterations. This allows us to express the CC at the targeted crash instruction through another condition CC' at the beginning of the controlling loop(s). Saxena et al. used abstract interpretation and pattern matching to infer the function [2]. In *Hercules*, we infer the function using data fitting on runtime values in registers and memory locations during loop exploration [3]. A similar idea has been successfully applied in the context of segmented symbolic analysis to discover symbolic relationships between program variables [4].

Figure 5 shows an example of loop-controlled crash instruction in a crash module *flvff.dll* that causes a memory access violation in *Real Player* due to an integer overflow vulnerability (`CVE-2010-3000`). In this example, the crash function is iteratively called in a loop and the crash instruction at `0x6116114a` attempts to store data to the targeted memory address that is calculated using the value of `eax` register. If the address is out of bounds, the crash will occur. Hence, the CC in this example must be expressed through symbolic data in `eax` as $eax == eax_{crash}$, where the eax_{crash} value comes from the register dump in the crash report. However, symbolic execution reaches the crash instruction with a concrete value in `eax` preventing the approach from formulating a symbolic crash condition.

We apply inter-procedural data flow analysis to establish whether the crash instruction is loop-controlled and, if so, to detect data dependencies between the operands of the instruction and the variables within the loop. In the example, we discover a data dependency between `eax` and `ecx` in the crash function (‘`mov eax, ecx`‘ at `0x61161146`) and between `ecx` and `esi` in the call site (‘`mov ecx, esi`‘ at `0x61161745`). Inside the loop, `esi` is incremented by a concrete value (passed through a function argument) at each iteration. The value of `eax` in crash instruction depends on the value of `esi` register inside the loop and in turn, the value of `esi` depends on the number of loop iterations.

Using data fitting, *Hercules* infers a relationship between `esi` and a loop count `it`: $\texttt{esi} = \texttt{esi}_0 + \texttt{it} \star \texttt{0x23}$. In the example, the number of loop iterations is controlled by the value of register `edi` that holds symbolic data (instructions at `0x6116174c` and `0x6116174d`). In other words, the value of `eax` in crash instruction is indirectly controlled by the symbolic input data in `edi`.

As a result, we transform the concrete constraint CC on `eax` at the crash instruction to a symbolic constraint CC' on the value of `edi` before the start of the loop. With this data we can synthesize crashing input by solving the formula $\phi' \wedge CC'$, where ϕ' is the path constraints to reach the loop.

B. Tackling path explosion

To avoid path explosion during concolic execution of real-world binaries our approach tackles its most prevalent sources – loops and string manipulation functions. To tackle path explosion in loops we bound a number of loop iterations in which concolic executor forks new feasible states. Beyond the bound the executor does not fork new states in a loop. Recent research shows that bounding loop iterations is a practical and effective solution in the context of symbolic execution [5].

String manipulation functions are more difficult to tackle than loops. In essence these functions are sophisticated loops over string data that are modelled with bit-vectors and processed as unbounded data causing generation of infinitely many symbolic states. Yet, symbolic exploration with string data is important, a large class of crashes in software is caused by buffer- and heap-overflows when programs operate on string data.

We define a heuristic that leverages string length estimation and approximation of standard string manipulation functions to help concolic execution in generating states with realistic string

data while reducing the risk of path explosion. The intuition behind our heuristic comes from the following observations: there are many concrete strings encoded in the program code and thus many string length bounds can be obtained based on operations between symbolic and concrete strings. Moreover, there are practical limitations on the size of strings such as function stack frame size and input file layout that provide estimates of string lengths. Finally, semantics of several standard string manipulation functions can be abstracted to the level of groups of paths and inform symbolic execution.

For functions like `strlen(sym)` we bound concolic exploration in the function according to the length estimate of a symbolic string parameter `sym` that we gather dynamically from a number of sources. A length estimate for strings allocated on stack should not exceed a current stack frame size, while file layout and object boundaries (boundaries between symbolic and concrete input data) indicate upper bounds for lengths of strings derived from input file data.

For other standard string functions that operate on pairs of strings we approximate these functions by mapping their few semantically different high-level paths to a multitude of low-level paths. One example of groups of high-level paths for a function `stricmp(str1,str2)` would be: (1) strings are equal, (2) strings are equal length and differ in content, and (3) strings differ in both length and content. These three groups map to thousands of feasible low-level paths stemming from two reasons. First, in LLVM based symbolic execution engine – *S2E* in our case – the string function is converted to LLVM bitcode that has larger number of branch instructions that in source code or binary. For `stricmp(str1,str2)` function the number of branches in LLVM bitcode is 13 versus 3 in source code. Second, the number of paths is also controlled by the number of loop iterations that depends on the length of the input strings. We define the high-level semantics of the string functions as a logical formula over function input, output and properties of the input, such as length of a string argument.

To avoid path explosion, during concolic execution we bound the exploration of these functions until paths from all semantically different path groups are generated, while controlling the number of generated paths. Consecutively, we prioritize groups of paths and select single paths from each group for further concolic exploration. For instance, for `stricmp()` function we give a higher priority to the path producing equal strings which covers the highly relevant case.

An experimentation with Orbital Viewer case study (`CVE-2010-0688`) highlights the degree of reduction in path numbers our technique achieves for concolically exploring a standard string function. *S2E* with depth-first search configuration would need to fork $(2^{13}) * 18 \approx 150$K paths to fully concolically explore `stricmp(str1,str2)` function with one symbolic string argument and one concrete string of length 18.

With our heuristic concolic executor only needs to explore 8K paths to populate elements for three high-level groups of total 19 paths that we keep: one path (strings are equal), one path (strings of equal length and differ in content), and 17 paths (strings differ in both length and content). Each path in the third group corresponds to the strings being unequal in any of the first 17 characters. We only need to keep 19 paths to cover all of the three high level paths of the `stricmp(str1,str2)` function, while the remaining low-level paths can be removed from exploration. Overall, we generate few paths that cover all high-level paths of a function in a balanced way and produce realistic strings.

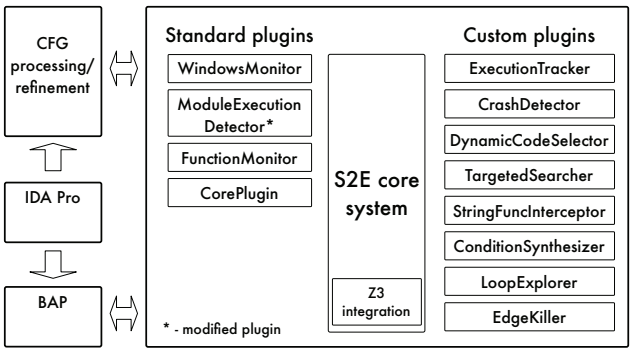

Fig. 6. Components of the *Hercules* toolset

VI. Implementation

Our approach *Hercules* builds upon and extends the selective symbolic execution technique *S2E* [1]. Figure 6 shows an overall view of the components of our toolset. The main components of our system are built as custom *S2E* plugins. In addition, *Hercules* provides tools for control flow graph processing outside *S2E* and data flow analysis built on *BAP*.

A. CFG refinement and path pruning functionality

Hercules implements analyses for post-processing the output of *IDA Pro* toolset and obtaining the static and the dynamic program structure information. We build CFG for each selected module of the system using the static program structure information (direct jumps, direct calls, jump tables) and the dynamic information (indirect register jumps and calls). We refine the CFG whenever the dynamic program structure is updated, while exploring the program under test in *Steps 2* and *3* of our approach.

A *PathPruner* module implements a pruning algorithm similar to the algorithm for computing "chop" by Brumley et al. [6]. *PathPruner* indicates every path that *does not* lead to interesting targets in a module dependency chain. In the crashing module, this tool will prune the paths that do not reach the crash location. The output of the tool will be used as the input of a plugin *EdgeKiller* that will kill a *S2E* state in runtime if it executes an undesirable path.

B. Extensions of the S2E core

STP, the SMT solver in *S2E*, does not compute `unsat_cores`. To get the `unsat_core` of a symbolic expression we integrate *Z3* with *S2E* and pass symbolic constraints between them in SMT2 format. Our framework augments *S2E* to output symbolic formulae in SMT2 format and implements a wrapper function to invoke *Z3* solver from *S2E*.

Another *S2E* core update takes snapshots of *S2E* states in the targeted search. We make snapshots of symbolic states at each branch location during concolic execution to enable backtracking of concolic executor. This functionality is implemented on top of cloning functionality of *KLEE* used by *S2E* and our version supports state cloning at an arbitrary execution point.

C. Analysis and search plugins

An *ExecutionTracker* plugin outputs important runtime information. It handles signals emitted by *S2E core* plugin when it executes an instruction or a basic block. In addition, it detects the Process_ID of the program under analysis to keep track of the information it produces and excludes information produced by other programs that use shared libraries.

A *DynamicCodeSelector* plugin enables flexible runtime selection of modules executed concolically (with forking enabled). The original *S2E* CodeSelector plugin is less flexible and only supports static configuration of a list of modules in which *S2E* selectively enables forking. Our *TargetedSearcher* plugin heavily relies on *DynamicCodeSelector* for dynamically switching different search stages each having different configurations of forking-enabled modules.

To synthesize crash conditions for loop-controlled crash instructions, we have developed three components. First, *ConditionSynthesizer* is built as *S2E* plugin. It outputs runtime values of all registers and updated variables at each iteration inside the controlling loop. Second, a light-weight data flow analysis is built on Binary Analysis Platform (*BAP*) [7]. Its output supports user in selecting registers/variables having relationship with a number of loop iterations. Third, a tool to interface with the *R* statistical package to invoke its regression models and infer function on dependent registers/variables and the number of loop iterations [8]. *Hercules* infers functions for simple and nested loops and covers three function forms – linear, polynomial, and exponential – by using simple linear and multiple linear regression models with logarithm and variable substitution transformations.

StringFunctionInterceptor controls the exploration inside string functions. It intercepts every call to the list of standard string library functions such as `strlen`, `strcpy`, `strcmp`, `stricmp`, `strcat`, `strchr` and `strstr` using handling signals emitted by the *FunctionMonitor* plugin of *S2E* (`onFunctionCall` and `onFunctionRet` signals). For each of the functions we implement a special structure to define groups of semantically distinct high-level paths (Section V-B). Each group is defined as a logical expression over function input, lengths of manipulated strings and function output. Finally, the module dynamically extracts the stack frame size of the caller to estimate string length bounds.

A *TargetedSearcher* plugin is a combination of the three searchers (1) *PathReplaySearcher*, (2) *SymbolicSummarization* and (3) *EntryPointTargetedSearcher*. Each searcher implements the dedicated phases of the targeted search algorithm defined in Section IV. The plugin switches between the searchers in the process of concolic execution using several signals emitted by the *S2E Core* plugin (`onStateFork`, `onStateSwitch`, `onExecuteInstruction`) and signals from our custom plugins. In particular, `onStringFunctionStart` and `onStringFunctionEnd` signals generated by the *StringFunctionInterceptor* plugin are used for state grouping and prioritization for string manipulation functions. *TargetedSearcher* populates the groups of states defined for each string function, prioritizes these states and removes redundant ones.

A *CrashDetector* module detects application crash by tracking Windows error reporting service invocation and calls *S2E* API to solve path constraint and generate crashing input.

VII. Experimental evaluation

We evaluated our approach experimentally on real-world application binaries. In this section we present the results of the evaluation that demonstrate that *Hercules* successfully reproduced *six* distinct crashes in five applications: Adobe Reader (AR), Windows Media Player (WMP), Real Player (RP), Orbital Viewer (OV) and Music Animation Machine (MAM) Player. Table I summarizes the results for the effectiveness of our approach as compared to the original *S2E* technique and widely used industrial black-box fuzzing tool *PeachFuzzer* (http://peachfuzzer.com). *Hercules* generated test inputs and reproduced all six crashes, whereas baseline techniques failed or took considerably more time to succeed.

A. Experimental setup

We conducted all of the experiments on a computer with a 3.4 GHz Intel Core i7-2600 CPU and 8 GB of RAM. The host OS is Ubuntu 12.04 64-bit. The guest OS are *Windows 7* Enterprise 32-bit SP1 and *Windows XP* 32-bit SP3. Our approach is implemented on *S2E* version from May 2, 2014 obtained at https://github.com/dslab-epfl/s2e. We used freeware *IDA Pro* 5.0 to disassemble binaries. In Table I, case studies marked with (*) have been tested on both Windows XP and Windows 7.

The case studies cover vulnerabilities of the four prevalent types (buffer overflow, integer overflow, memory access violation and division-by-zero) from http://cve.mitre.org/ and operate on five distinct structured file formats. For the OV case study we used a developer test suite obtained at http://www.orbitals.com/orb/ov.htm. For the Adobe Reader case study, we used Microsoft Word 2010 to create pdf files with embedded fonts. For the other four case studies, we obtained test suites on the Internet from a random sample of benign files of an appropriate format. Table I highlights the test suite composition with numbers of benign files and their variation in size. We enabled forking in a subset of modules indicated by the crash reports as shown in Table I.

The toolset was configured for a timeout after twelve hours of exploration and run without parallelization of the execution process. For *Hercules*, we have fixed a loop bound of three iterations and state timeout of 30 seconds to prevent the exploration from "drifting" (Section III).

Table I shows execution times for the CFG construction (*Step 1*), concolic (*Step 2*) and the targeted (*Step 3*) exploration by *Hercules* as per Figure 1 (correspondingly marked ①,②,③

TABLE I
EXPERIMENTAL SETUP AND RESULTS

Vulnerability	CVE-2014-2671	CVE-2010-0718	CVE-2010-0688	CVE-2011-0502	CVE-2010-2204	CVE-2010-3000
Application	WMP v9.0	WMP v9.0	OV v1.04	MAM v0.35	AR v9.2	RP SP 1.0
Selected / total modules	4 / 84	3 / 86	2 / 49	1 / 51	2/78	2/129
Size of crash. module	1.22 MB	1.22 MB	538 KB	368 KB	2.32 MB	60 KB
Test suite – No. of files	10 (2–137 KB)	15 (2–54 KB)	10 (3–5 KB)	10 (2–5 KB)	5 (55–307 KB)	6 (87–654 KB)
S2E (Random search)	NO (>12 hr)	NO (>12 hr)	NO (>12 hr)	YES (2 min)	NO (>12 hr)	NO (>12 hr)
S2E (DFS search)	NO (>12 hr)	NO (>12 hr)	NO (out of mem.)	NO (>12 hr)	NO (>12 hr)	NO (>12 hr)
PeachFuzzer	NO (>24 hr)	NO (>24 hr)	YES (10 hr)	YES (10 min)	NO (>24 hr)	NO (>24 hr)
Hercules ① + ②	5 min + 45 min	5 min + 1 hr 30 min	2 min + 2 hr	1 min + ∼0 sec	5 min + 2 hr	5 min + 2 hr
Hercules ③	YES (15 min)	YES (60 min)	YES (40 min)	YES (30 sec)*	YES (60 min)*	YES (45 min)

in the table). For *Step 1*, our automated scripts construct CFG from the output of *IDA Pro* within few minutes. We used a practical time limit of two hours for exploration in *Step 2*. For the five case studies (except CVE-2011-0502), *Hercules* (*Step 2*) explored, resolved dynamic information and reached a crashing module within two hours, while the case study on MAM (CVE-2011-0502) did not require exploration phase to reach the crashing module. Finally, for all the case studies targeted search (*Step 3*) reproduced the crashes within an hour.

B. Reproducing crashes

Our approach reached and reproduced crashes CVE-2014-2671 and CVE-2010-0718 in Windows Media Player (Quartz library). CVE-2014-2671 is a vulnerability in Windows Media Player version 9. Attackers can exploit this vulnerability to cause a denial of service via a crafted `.wav` file. CVE-2010-0718 is a buffer overflow vulnerability in Windows Media Player version 9. It allows attackers to cause a denial of service via a crafted `.mpg` or `.mid` file that triggers a system crash due to a divide-by-zero exception. *Hercules* successfully reproduced the two crashes using the targeted search. In both cases, *Hercules* avoided state explosion by bounding loop iterations, while no string function analysis was required. *Hercules* reproduced CVE-2010-0718 using as little as 1% of input data in symbolic form.

CVE-2010-0688 is a crucial stack-based overflow in Orbital Viewer, a tool for visualization of atomic and molecular orbitals. By using a crafted `.orb` or `.ov` file, attackers can trigger a system crash in Memory Access Violation exception or execute arbitrary code. The vulnerability comes from the code for reading data from input file using a known vulnerable function `fscanf`. OV does not correctly check the data size before writing it into stack buffers. The crash happens when the overwritten data section is accessed by OV after a series of function calls, including calls to string manipulation functions. *Hercules* successfully bridged the distance between the location where crashing data is introduced and the crashing location by leveraging our heuristic for exploring string functions (Section V-B), and reproduced the crash.

CVE-2011-0502 is a vulnerability in MAM MIDI Player that allows attackers to easily cause a denial of service via a crafted `.mid` file that crashes the program with a null pointer dereference. This is the most "simple" case study in our experiments that *Hercules* reproduced within 30 seconds. Furthermore, *Hercules* does not require loop bounding nor string function analysis to reproduce the crash.

CVE-2010-2204 is an vulnerability in Adobe Reader 9.0–9.3 that allows attackers to cause a denial of service or execute arbitrary code. CVE-2010-3000 is an integer overflow vulnerability in RealPlayer SP 1.0 that allows attackers to execute arbitrary code. For both cases *Hercules* can reach crash instructions by symbolically executing the programs with benign inputs, however, the programs do not crash, because the crash instructions are loop-controlled. With the loop-controlled crash condition analysis (Section V-A) *Hercules* can identify the loop and infer a relationship between crash instructions and the controlling loops. As a result, *Hercules* can successfully synthesize symbolic crash conditions on the number of loop iterations and use them to reproduce both crashes.

C. Comparing with the baseline

We demonstrate the effectiveness of *Hercules* by comparing it with the baseline *S2E* and black-box fuzzing tool *PeachFuzzer* on the same six case studies. We have run *S2E* with the input files that *Hercules* used to successfully reproduce the crashes, while *PeachFuzzer* used all the files in each test suite.

The results shown in Table I demonstrate that *S2E* can reproduce the "simple" crash (CVE-2011-0502) and fails to reproduce the other ones. Non-directed search of the baseline *S2E* prevents it from reaching relevant program locations in a given time and state space constraints. When run with a depth first search (DFS) exploration, *S2E* digs itself in a single path, while for the OV case study (CVE-2010-0688) it gets path explosion in string manipulation functions before reaching the crash location.

We run *PeachFuzzer* in a fully automatic setting with infinite iterations of random mutation strategy and without user-provided data model (input grammar specification) for up to 24 hours. *PeachFuzzer* took substantially more time than *Hercules* to generate crashing inputs for two case studies. Effectiveness of the fuzzing tool critically depends on the results of manual analysis to provide it with a correct input grammar specification and indicate input portions that can and must be mutated, and portions that need to be preserved.

VIII. RELATED WORK

Recent research produced a rich body of work for intelligent navigation of program space and assisting symbolic execution

in reaching certain program locations [9]–[15]. A common line in these approaches is to use distance metrics and anchor points for driving symbolic execution.

Debugging approach *ESD* [9] and patch testing approach *KATCH* [11] share the goal of our approach to systematically direct symbolic exploration towards a specific program location. Differently from our approach that works with binaries and partially resolved system information, *ESD* and *KATCH* assume availability of a program source code and thus have more precise system information including inter-procedural CFG to inform the techniques and apply data flow analysis. Both approaches analyze data-flows to identify reaching definitions responsible for taking critical control-dependent edges and steer symbolic execution towards these intermediate goals using proximity metric.

Approaches that work on program binaries focus on resolving sufficient system information using static and dynamic analyses [12], [16], [17]. Approach by Babić et al. uses static analysis to guide automated dynamic test generation [16]. Dynamic analysis resolves indirect jumps with seed tests, and the static analysis helps symbolic execution directing exploration towards vulnerabilities based on the shortest paths and loop pattern heuristics. *MACE* by Cho et al. combines symbolic and concrete execution to build and refine an abstract finite state model of the system-environment interaction and use it to guide the program exploration [12]. *HI-CFG* by Caselden et al. generates hybrid information- and control-flow graph of a program to direct stages of backwards symbolic execution. Analogously to these approaches, *Steps 1* and *2* of our approach resolve program structure information and use it to inform symbolic execution.

Fuzzing techniques integrate and interleave with symbolic and concolic approaches for synthesizing complex structured program inputs [15], [18]–[20]. A notable example of white-box fuzzing technique is *SAGE* [18]. It works on binaries and given an initial test input concolically explores the application using code-coverage maximizing heuristic. *BuzzFuzz*, an automated source code level technique, uses dynamic taint tracing to automatically locate regions of given seed input files that influence values used at program attack points and infers the type of the input based on the taint information [20]. *Dowser* uses taint analysis to identify program inputs that influence memory accesses and uses concolic execution with partially symbolic inputs for learning about pointer access patterns [15]. Using this information, *Dowser* steers fuzzing technique towards complex pointer calculations in the program. Likewise, by leveraging benign input files for generating hybrid symbolic inputs, our approach retains intrinsic input structure helping to scale symbolic exploration.

The idea of summarizing functions or problematic behavior in symbolic execution have been investigated earlier. Godefroid proposed a compositional approach to capture and reuse function summaries to scale dynamic symbolic execution [21]. Brumley et al. describe vulnerability signatures as weakest preconditions [6]. Several approaches have explored similar intuition for summarizing and reasoning about problematic behavior in a backwards fashion to find program inputs that trigger such behavior [13], [17], [22], [23].

A related line of research improves the scalability of symbolic execution [2], [4], [14], [24]–[27]. Kuznetsov et al. introduced dynamic state merging and query count estimation [25]. By estimating the impact of symbolic variables on solver queries their approach merges states balancing between the number of generated states and the complexity of the queries to the solver. *Mayhem* by Cha et al. combines online and offline symbolic execution and models symbolic memory at the binary level [24]. Built on *Mayhem*, *Veritesting* enhances dynamic symbolic execution with static symbolic execution [26].

These approaches are orthogonal to our work and can be integrated to enhance its scalability. Our approach deals with high-level sources of scalability issues in symbolic execution that are manifested in string manipulation functions. Prior research has been addressing the issues of operating on string data in dynamic analysis and symbolic execution [28]–[31]. Larson and Austin characterize and track bounds and null termination of string variables for dynamically checking validity of program inputs [28]. Xu et al. track the abstract length of the input string prefixes and instrument the string library to update abstract lengths in symbolic execution [29]. Bucur et al. associate high-level execution paths of the program to some low-level execution paths during symbolic execution of *python* programs [31]. Our approach learns from these ideas and extends them in a new context of analyzing real-world application binaries. In our future work we plan to enhance our technique by integrating string solvers that can be adapted for working with program binaries [30].

IX. Discussion

In this paper, we have presented the design and evaluation of our *Hercules* approach for finding test inputs which can reproduce a given crash. Our approach is based on symbolic execution and its distinctive features include (i) working on binaries without source code and encompassing techniques to construct the control-flow graph directly from binaries in the presence of register-indirect jump instructions, (ii) combining taint tracking and symbolic execution to find which parts of the input file must be kept symbolic, and (iii) search strategies to direct a path towards the crashing location by analyzing why the current path being traversed by the search cannot reach the crash. Experiments on real-world application binaries such as Windows Media Player and Adobe Reader, show the efficacy of our approach in finding test inputs to reproduce a crash.

Acknowledgment

This research is supported in part by a grant from DSO National Laboratories, Singapore, and in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. The authors would like to thank Kenneth Cheong and Chia Yuan Cho for their constructive comments at various stages of the work.

REFERENCES

[1] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. ACM, 2011, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950396

[2] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572299

[3] D. Freedman, *Statistical models : theory and practice*. Cambridge New York: Cambridge University Press, 2009.

[4] W. Le, "Segmented symbolic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 212–221. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486817

[5] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, 2013, pp. 246–256.

[6] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, July 2007, pp. 311–325.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, Jul. 2011.

[8] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[9] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. ACM, 2010, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755946

[10] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 474–484. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337279

[11] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491438

[12] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery," in *Proceedings of the 20th USENIX Security Symposium*, Aug 2011.

[13] K.-K. Ma, K. Yit Phang, J. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis*, ser. Lecture Notes in Computer Science, E. Yahav, Ed. Springer Berlin Heidelberg, 2011, vol. 6887, pp. 95–111. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23702-7_11

[14] J. Jaffar, V. Murali, and J. A. Navas, "Boosting concolic testing via interpolation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 48–58. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491425

[15] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller

[16] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. ACM, 2011, pp. 12–22. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001423

[17] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "Hi-cfg: Construction by binary analysis and application to attack polymorphism," in *Computer Security – ESORICS 2013*, ser. Lecture Notes in Computer Science, J. Crampton, S. Jajodia, and K. Mayes, Eds. Springer Berlin Heidelberg, 2013, vol. 8134, pp. 164–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40203-6_10

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Network and Distributed System Security Symposium*, 2008.

[19] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[20] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 474–484. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070546

[21] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. ACM, 2007, pp. 47–54. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190226

[22] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[23] C. Y. Cho, V. D'Silva, and D. Song, "Blitz: Compositional bounded model checking for real-world programs," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 136–146.

[24] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254088

[26] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. 36th International Conference on Software Engineering*, Jun. 2014.

[27] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *ECOOP*, 2008, pp. 542–565.

[28] E. Larson and T. Austin, "High coverage detection of input-related security faults," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, 2003, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251362

[29] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390636

[30] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 114–124. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491456

[31] S. Bucur, J. Kinder, and G. Candea, "Prototyping Symbolic Execution Engines for Interpreted Languages," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.