CRETE: A Versatile Binary-Level Concolic Testing Framework

Bo Chen[1(✉)], Christopher Havlicek[2], Zhenkun Yang[2], Kai Cong[2], Raghudeep Kannavara[2], and Fei Xie[1]

[1] Portland State University, Portland, OR 97201, USA
{chenbo,xie}@pdx.edu
[2] Intel Corporation, Hillsboro, OR 97124, USA
{christopher.havlicek,zhenkun.yang,kai.cong, raghudeep.kannavara}@intel.com

Abstract. In this paper, we present CRETE, a versatile binary-level concolic testing framework, which features an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution engine backends. CRETE's extensibility is rooted in its modular design where concrete and symbolic execution is loosely coupled only through standardized execution traces and test cases. The standardized execution traces are LLVM-based, self-contained, and composable, providing succinct and sufficient information for symbolic execution engines to reproduce the concrete executions. We have implemented CRETE with KLEE as the symbolic execution engine and multiple concrete execution frontends such as QEMU and 8051 Emulator. We have evaluated the effectiveness of CRETE on GNU COREUTILS programs and TianoCore utility programs for UEFI BIOS. The evaluation of COREUTILS programs shows that CRETE achieved comparable code coverage as KLEE directly analyzing the source code of COREUTILS and generally outperformed ANGR. The evaluation of TianoCore utility programs found numerous exploitable bugs that were previously unreported.

1 Introduction

Symbolic execution [1] has become an increasingly important technique for automated software analysis, e.g., generating test cases, finding bugs, and detecting security vulnerabilities [2–11]. There have been many recent approaches to symbolic execution [12–22]. Generally speaking, these approaches can be classified into two categories: online symbolic execution (e.g., BitBlaze [4], KLEE [5], and S^{2}E [6]), and concolic execution (a.k.a., offline symbolic execution, e.g., CUTE [2], DART [3], and SAGE [7]). Online symbolic execution closely couples Symbolic Execution Engines (SEE) with the System Under Test (SUT) and explore all possible execution paths of SUT online at once. On the other hand, concolic execution decouples SEE from the SUT through traces, which concretely runs a single execution path of a SUT and then symbolically executes it.

© The Author(s) 2018
A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 281–298, 2018.
https://doi.org/10.1007/978-3-319-89363-1_16

Both online and offline symbolic execution are facing new challenges, as computer software is experiencing an explosive growth, both in complexities and diversities, ushered in by the proliferation of cloud computing, mobile computing, and Internet of Things. Two major challenges are: (1) the SUT involves many types of software for different hardware platforms and (2) the SUT involves many components distributed on different machines and as a whole the SUT cannot fit in any SEE. In this paper, we focus on how to extend concolic execution to satisfy the needs for analyzing emerging software systems. There are two major observations behind our efforts on extending concolic execution:

– The decoupled architecture of concolic execution provides the flexibility in integrating new trace-captured frontends for emerging platforms.
– The trace-based nature of concolic testing offers opportunities for selectively capturing and synthesizing reduced system-level traces for scalable analysis.

We present CRETE, a versatile binary-level concolic testing framework, which features an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution backends. CRETE's extensibility is rooted in its modular design where concrete and symbolic execution is loosely coupled only through standardized execution traces and test cases. The standardized execution traces are LLVM-based, self-contained, and composable, providing succinct and sufficient information for SEE to reproduce the concrete executions. The CRETE framework is composed of:

– **A CRETE tracing plugin,** which is embedded in the concrete execution environment, captures binary-level execution traces of the SUT, and stores the traces in a standardized trace format.
– **A CRETE manager,** which archives the captured execution traces and test cases, schedules concrete and symbolic execution, and implements policies for selecting the traces and test cases to be analyzed and explored next.
– **A CRETE replayer,** which is embedded in the symbolic execution environment, performs concolic execution on captured traces for test case generation.

We have implemented the CRETE framework on top of QEMU [23] and KLEE, particularly the tracing plugin for QEMU, the replayer for KLEE, and the manager that coordinates QEMU and KLEE to exchange runtime traces and test cases and manages the policies for prioritizing runtime traces and test cases. To validate CRETE extensibility, we have also implemented a tracing plugin for the 8051 emulator [24]. The trace-based architecture of CRETE has enabled us to integrate such tracing frontends seamlessly. To demonstrate its effectiveness and capability, we evaluated CRETE on GNU COREUTILS programs and TianoCore utility programs for UEFI BIOS, and compared with KLEE and ANGR, which are two state-of-art open-source symbolic executors for automated program analysis at source-level and binary-level.

The CRETE framework makes several key contributions:

– **Versatile concolic testing.** CRETE provides an open and highly extensible architecture allowing easy integration of different concrete and symbolic execution environments, which communicate with each other only by exchanging

standardized traces and test cases. This significantly improves applicability and flexibility of concolic execution to emerging platforms and is amenable to leveraging new advancements in symbolic execution.

- **Standardizing runtime traces.** CRETE defines a standard binary-level trace format, which is LLVM based, self-contained and composable. Such a trace is captured during concrete execution, representing an execution path of a SUT. It contains succinct and sufficient information for reproducing the execution path in other program analysis environment, such as for symbolic execution. Having standardized traces minimizes the need of converting traces for different analysis environment and provides a basis for common trace-related optimizations.
- **Implemented a CRETE prototype.** We have implemented CRETE with KLEE as the SEE backend and multiple concrete execution frontends such as QEMU and 8051 Emulator. CRETE achieved comparable code coverage on COREUTILS binaries as KLEE directly analyzing at source-level and generally outperformed ANGR. CRETE also found 84 distinct and previously-unreported crashes on widely-used and extensively-tested utility programs for UEFI BIOS development. We also make CRETE implementation publicly available to the community at github.com/SVL-PSU/crete-dev.

2 Related Work

DART [3] and CUTE [2] are both early representative work on concolic testing. They operate on the source code level. CRETE further extends concolic testing and targets close-source binary programs. SAGE [7] is a Microsoft internal concolic testing tool that particularly targets at X86 binaries on Windows. CRETE is platform agnostic: as long as a trace from concrete execution can be converted into the LLVM-based trace format, it can be analyzed to generate test cases.

KLEE [5] is a source-level symbolic executor built on the LLVM infrastructure [25] and is capable of generating high-coverage test cases for C programs. CRETE adopts KLEE as its SEE, and extends it to perform concolic execution on standardized binary-level traces. S^2E [6] provides a framework for developing tools for analyzing close-source software programs. It augments a Virtual Machine (VM) with a SEE and path analyzers. It features a tight coupling of concrete and symbolic execution. CRETE takes a loosely coupled approach to the interaction of concrete and symbolic execution. CRETE captures complete execution traces of the SUT online and conducts whole trace symbolic analysis off-line.

BitBlaze [4] is an early representative work on binary analysis for computer security. It and its follow-up work Mayhem [8] and MergePoint [12] focus on optimizing the close coupling of concrete and symbolic execution to improve the effectiveness in detecting exploitable software bugs. CRETE has a different focus on providing an open architecture for binary-level concolic testing that enables flexible integration of various concrete and symbolic execution environments.

ANGR [14] is an extensible Python framework for binary analysis using VEX [26] as an intermediate representation (IR). It implemented a number of

existing analysis techniques and enabled the comparison of different techniques in a single platform. ANGR needs to load a SUT in its own virtual environment for analysis, so it has to model the real execution environment for the SUT, like system calls and common library functions. CRETE, however, performs in-vivo binary analysis, by analyzing binary-level trace captured from unmodified execution environment of a SUT. Also, ANGR needs to maintain execution states for all paths being explored at once, while CRETE reduces memory usage dramatically by analyzing a SUT path by path and separates symbolic execution from tracing.

Our work is also related to fuzz testing [27]. A popular representative tool for fuzzing is AFL [28]. Fuzzing is fast and quite effective for bug detection; however, it can easily get stuck when a specific input, like magic number, is required to pass a check and explore new paths of a program. Concolic testing guides the generation of test cases by solving constraints from the source code or binary execution traces and is quite effective in generating complicated inputs. Therefore, fuzzing and concolic testing are complementary software testing techniques.

3 Overview

During the design of the CRETE framework for binary-level concolic testing, we have identified the following design goals:

- **Binary-level In-vivo Analysis.** It should require only the binary of the SUT and perform analysis in its real execution environment.
- **Extensibility.** It should allow easy integration of concrete execution frontends and SEE backends.
- **High Coverage.** It should achieve coverage that is not significantly lower than the coverage attainable by source-level analysis.
- **Minimal Changes to Existing Testing Processes.** It should simply provide additional test cases that can be plugged into existing testing processes without major changes to the testing processes.

To achieve the goals above, we adopts an online/offline approach to concolic testing in the design of the CRETE framework:

- **Online Tracing.** As the SUT is concretely executed in a virtual or physical machine, an online tracing plugin captures the binary-level execution trace into a trace file.
- **Offline Test Generation.** An offline SEE takes the trace as input, injects symbolic values and generates test cases. The new test cases are in turn applied to the SUT in the concrete execution.

This online tracing and offline test generation process is iterative: it repeats until all generated test cases are issued or time bounds are reached. We extend this process to satisfy our design goals as follows.

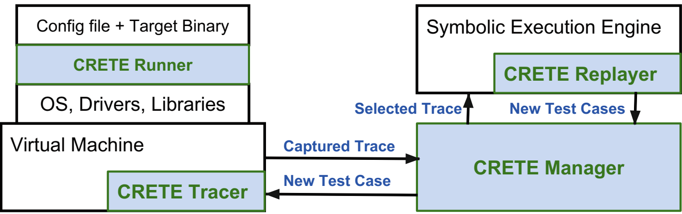

Fig. 1. CRETE architecture

- Execution traces of a SUT are captured in its unmodified execution environment on binary-level. The tracing plugin can be an extension into a VM (Sect. 4.1), a hardware tracing facility, or a dynamic binary instrumentation tool, such as PIN [29], and DynamoRIO [30].
- The concrete and symbolic execution environments are decoupled by standardized traces (Sect. 4.2). As long as they can generate and consume standardized traces, they can work together as a cohesive concolic process.
- Optimization can be explored on both tracing and test case generation, for example, selective binary-level tracing to improve scalability (Sect. 4.3), and concolic test generation to reduce test case redundancy (Sect. 4.4). This makes high-coverage test generation on binary-level possible.
- The tracing plugin is transparent to existing testing processes, as it only collects information. Therefore, no change is made to the testing processes.

4 Design

In this section, we present the design of CRETE with a VM as the concrete execution environment. The reason for selecting a VM is that it allows complete access to the whole system for tracing runtime execution states and is generally accessible as mature open-source projects.

4.1 CRETE Architecture

As shown in Fig. 1, CRETE has four key components: **CRETE Runner**, a tiny helper program executing in the guest OS of the VM, which parses the configuration file and launches the target binary program (TBP) with the configuration and test cases; **CRETE Tracer**, a comprehensive tracing plug-in in the VM, which captures binary-level traces from the concrete execution of the TBP in the VM; **CRETE Replayer**, an extension of the SEE, which enables the SEE to perform concolic execution on the captured traces and to generate test cases; **CRETE Manager**, a coordinator that integrates the VM and SEE, which manages runtime traces captured and test cases generated, coordinates the concrete and symbolic execution in the VM and the SEE, and iteratively explores the TBP.

CRETE takes a TBP and a configuration file as inputs, and outputs generated test cases along with a report of detected bugs. The manual effort and learning

curve to utilize CRETE are minimal. It makes virtually no difference for users to setup the testing environment for the TBP in a CRETE instrumented VM than a vanilla VM. The configuration file is an interface for users to configure parameters on testing a TBP, especially specifying the number and size of symbolic command-line inputs and symbolic files for test case generation.

4.2 Standardized Runtime Trace

To enable the modular and plug-and-play design of CRETE, a standardized binary-level runtime trace format is needed. A trace in this format must capture sufficient information from the concrete execution, so the trace can be faithfully replayed within the SEE. In order to integrate a concrete execution environment to the CRETE framework, only a plug-in for the environment needs to be developed, so that the concrete execution trace can be stored in the standard file format. Similarly, in order to integrate a SEE into CRETE, the engine only needs to be adapted to consume trace files in that format.

We define the standardized runtime trace format based on the LLVM assembly language [31]. The reasons for selecting the LLVM instruction sets are: (1) it has become a de-facto standard for compiler design and program analysis [25,32]; (2) there have been many program analysis tools based on LLVM assembly language [5,33–35]. A standardized binary-level runtime trace is packed as a self-contained LLVM module that is directly consumable by a LLVM interpreter. It is composed of (1) a set of assembly-level basic blocks in the format of LLVM functions (2) a set of hardware states in the format of LLVM global variables (3) a set of CRETE-defined helper functions in LLVM assembly (4) a main function in LLVM assembly. The set of assembly-level basic blocks is captured from a concrete execution of a TBP. It is normally translated from another format (such as QEMU-IR) into LLVM assembly, and each basic block is packed as a LLVM function. The set of hardware states are runtime states along the execution of the TBP. It consist of CPU states, memory states and maybe states of other hardware components, which are packed as LLVM global variables. The set of helper functions are provided by CRETE to correlate captured hardware states with captured basic blocks, and open interface to SEE. The main function represents the concrete execution path of the TBP. It contains a sequence of calls to captured basic blocks (LLVM functions), and calls to CRETE-defined helper functions with appropriate hardware states (LLVM global variables).

An example of a standardized runtime trace of CRETE is listed in Fig. 2. The first column of this figure is a complete execution path of a program with given concrete inputs. It is in the format of assembly-level pseudo-code. Assuming the basic blocks `BB_1` and `BB_3` are of interest and are captured by CRETE Tracer, while other basic blocks are not (see Sect. 4.3 for details). As shown in the second and third column of the figure, hardware states are captured in two categories, initial state and side-effects from basic blocks not being captured. As shown in the forth column of the figure, captured basic blocks are packed as LLVM functions, and captured hardware states are packed as LLVM global variables in

Concrete Execution Path			Initial HW State		HW Side Effects	
			CPU	Memory	CPU	Memory
			r0,r1,...,rn			
1	mem_ld r1, [0x1234]	BB_1		[0x1234]		
2	add r1, r0					
3	mem_st [0x1234], r1					
4	Br r1, inst_5, xxx					
5	mem_ld r1, [0x5678]	BB_2			r1	
6	add r1, r0				r1	
7	mem_st [0x5678], r1					[0x5678]
8	Jump inst_9					
9	mem_ld r0, [0x1234]	BB_3		[0x1234]		
10	add r1, r0					
11	mem_st [0x5678], r1					
12	Br r0, xxx, inst_13					
13	nop	BB_4				
14	nop					

Standardized Trace as a LLVM Module

```
@init_state = {<r0,r1,...,rn>, <[0x1234]>}
@side_effects = {<r1>,<[0x5678]>}

define asm_BB_1() {
 %1 = load * 0x1234
 %2 = getelementptr @init_state, r0_offset
 %3 = load * %2
 %4 = add %1, %3
 %5 = getelementptr @init_state, r1_offset
 store %4, * %5
 store %4, * 0x1234
 br %4, %.path_true, %.path_false
 %.path_true:
 %.path_false:
} ;asm_BB_3() is omitted

external sync_state() ;crete helper function

define main() {
 call sync_state(@init_state)
 call asm_BB_1()
 call sync_state(@side_effects)
 call asm_BB_3()
}
```

Fig. 2. Example of standardized runtime trace

the standardized trace. A main function is also added making the trace a self-contained LLVM module. The main function first invokes CRETE helper functions to initialize hardware states, then it calls into the first basic block LLVM function. Before it calls into the second basic block LLVM function, the main function invokes CRETE helper functions to update hardware states. For example, before calling `asm_BB_3`, it calls function `sync_state` to update register `r1` and memory location `0x5678`, which are the side effects brought by `BB_2`.

4.3 Selective Binary-Level Tracing

A major part of a standardized trace is assembly-level basic blocks which are essentially binary-level instruction sequences representing a concrete execution of a TBP. It is challenging and also unnecessary to capture the complete execution of a TBP. First, software binaries can be very complex. If we capture the complete execution, the trace file can be prohibitively large and difficult for the SEE to consume and analyze. Second, as the TBP is executing, it is very common to invoke many runtime libraries (such as `libc`) of no interest to testers. Therefore, an automated way of selecting the code of interest is needed.

CRETE utilizes Dynamic Taint Analysis (DTA) [36] to achieve selective tracing. The DTA algorithm is a part of CRETE Tracer. It tracks the propagation of tainted values, normally specified by users, during the execution of a program. It works on binary-level and in byte-wise granularity. By utilizing the DTA algorithm, CRETE Tracer only captures basic blocks that operate on tainted values, while only capturing side-effects from other basic blocks. For the example trace in Fig. 2, if the tainted value is from user's input to the program and is stored at memory location `0x1234`, DTA captures basic block `BB_1` and `BB_3`, because both of them operate on tainted values, while the other two basic blocks do not touch tainted values, and are not captured by DTA.

CRETE Tracer captures the initial state of CPU by capturing a copy of the CPU state before the first interested basic block is executed. The initial CPU state is normally a set of register values. As shown in Fig. 2, the initial CPU state is captured before instruction (1). Näively, the initial memory state can be captured in the same way; however, the typical size of memory makes it impractical to dump entirely. To minimize the trace size, CRETE Tracer only captures the parts of memory that are accessed by the captured read instructions, like instruction (1) and (9). The memory being touched by the captured write instructions, like instruction (3) and (11), can be ignored because the state of this part of the memory has been included in the write instructions and has been captured. As a result, CRETE Tracer monitors every memory read instruction that is of interest, capturing memory as needed on-the-fly. In the example above, there are two memory read instructions. CRETE Tracer monitors both of them, but only keeps the memory state taken from instruction (1) as a part of the initial state of memory, because instruction (1) and (9) access the same address.

The side effects of hardware states are captured by monitoring uncaptured write instructions of hardware states. In the example in Fig. 2, instructions (5) and (6) write CPU registers which cause side effects to the CPU state. CRETE Tracer monitors those instructions and keeps the updated register values as part of the runtime trace. As register `r1` is updated twice by two instructions, only the last update is kept in the runtime trace. Similarly, CRETE Tracer captures the side effect of memory at address `0x5678` by monitoring instruction (7).

4.4 Concolic Test Case Generation

While a standardized trace is a self-contained LLVM module and can be directly executed by a LLVM interpreter, it opens interfaces to SEE to inject symbolic values for test case generation. Normally SEE injects symbolic values by making a variable in source code symbolic. From source code level to machine code level, references of variables by names have become memory accesses by addresses. For instance, a reference of a concrete input variable of a program becomes a access of a piece of memory that stores the state of that input variable. CRETE injects self-defined helper function, `crete_make_concolic`, to the captured basic blocks while capturing trace. This helper function provides the address and size of the piece of memory for injecting symbolic values, along with a name to offer better readability for test case generation. By catching this helper function, SEE can introduce symbolic values at the right time and right place.

A standardized trace in CRETE represents only a single path of a TBP as shown in Fig. 3(a). Test case generation on this trace with näive symbolic execution by SEE won't be effective, as it ignores the single path nature of the trace. As illustrated in Fig. 3(b), native symbolic replay of CRETE trace produces execution states and test cases that are exponential to the number of branches within the trace. As shown in Fig. 3(c), with concolic replay of CRETE trace, the SEE in CRETE maintains only one execution state, requiring minimal memory usage, and generates a more compact set of test cases, whose number is linear to the number of branches in that trace. For a branch instruction in a captured basic

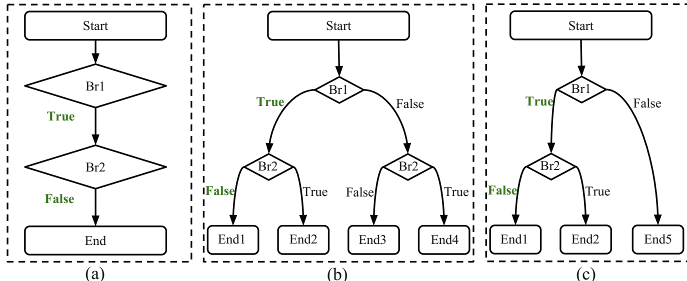

Fig. 3. Execution tree of the example trace from Fig. 2: (a) for concrete execution, (b) for symbolic execution, and (c) for concolic execution.

block, if both of the paths are feasible given the collected constraints so far on the symbolic values, the SEE in CRETE only keeps the execution state of the path that was taken by the original concrete execution in the VM by adding the corresponding constraints of this branch instruction, while generating a test case for the other path by resolving constraints with the negated branch condition. This generated test case can lead the TBP to a different execution path later during the concrete execution in the VM.

4.5 Bug and Runtime Vulnerability Detection

CRETE detects bugs and runtime vulnerabilities in two ways. First, all the native checks embedded in SEE are checked during the symbolic replay over the trace captured from concrete execution. If there is a violation to a check, a bug report is generated and associated with the test case that is used in the VM to generate this trace. Second, since CRETE does not change the native testing process and simply provides additional test cases that can be applied in the native process, all the bugs and vulnerability checks that are used in the native process are effective in detecting bugs and vulnerabilities that can be triggered by the CRETE generated test cases. For instance, Valgrind [26] can be utilized to detect memory related bugs and vulnerabilities along the paths explored by CRETE test cases.

5 Implementation

To demonstrate the practicality of CRETE, we have implemented its complete workflow with QEMU [23] as the frontend and KLEE [5] as the backend respectively. And to demonstrate the extensibility of CRETE, we have also developed the tracing plug-in for the 8051 emulator which readily replaces QEMU.

CRETE Tracer for QEMU: To give CRETE the best potential of supporting various guest platforms supported by QEMU, CRETE Tracer captures the basic blocks in the format of QEMU-IR. To convert captured basic blocks into standardized

trace format, we implemented a QEMU-IR to LLVM translator based on the x86-LLVM translator of S^2E [37]. We offload this translation from the runtime tracing as a separate offline process to reduce the runtime overhead of CRETE Tracer. QEMU maintains its own virtual states to emulate physical hardware state of a guest platform. For example, it utilizes virtual memory state and virtual CPU state to emulate states of physical memory and CPU. Those virtual states of QEMU are essentially source-level structs. CRETE Tracer captures hardware states by monitoring the runtime values of those structs maintained by QEMU. QEMU emulates the hardware operations by manipulating those virtual states through corresponding helper functions defined in QEMU. CRETE Tracer captures the side effects on those virtual hardware states by monitoring the invocation of those helper functions. As a result, the initial hardware states being captured are the runtime values of these QEMU structs, and the side effects being captured are the side effects on those structs from the uncaptured instructions.

CRETE Replayer for KLEE: KLEE takes as input the LLVM modules compiled from C source code. As the CRETE trace is a self-contained LLVM module, CRETE Replayer mainly injects symbolic values and achieves concolic test generation. To inject symbolic values, CRETE Replayer provides a special function handler for CRETE interface function `crete_make_concolic`. KLEE is an online symbolic executor natively, which forks execution states on each feasible branches and explores all execution paths by maintaining multiple execution states simultaneously. To achieve concolic test generation, CRETE Replayer extends KLEE to generate test cases only for feasible branches while not forking states.

CRETE Tracer for 8051 Emulator: The 8051 emulator executes a 8051 binary directly by interpreting its instructions sequentially. For each type of instruction, the emulator provides a helper function. Interpreting an instruction entails calling this function to compute and change the relevant registers and memory states. The tracing plug-in for the 8051 emulator extends the interpreter. When the interpreter executes an instruction, an LLVM call to its corresponding helper function is put in the runtime trace. The 8051 instruction-processing helper functions are compiled into LLVM and incorporated into the runtime trace serving as the helper functions that map the captured instructions to the captured runtime states. The initial runtime state is captured from the 8051 emulator before the first instruction is executed. The resulting trace is of the same format as that from QEMU and is readily consumable by KLEE.

6 Evaluation

In this section, we present the evaluation results of CRETE from its application to GNU COREUTILS [38] and TianoCore utility programs for UEFI BIOS [39]. Those evaluations demonstrate that CRETE generates effective test cases that are as effective in achieving high code coverage as the state-of-the-art tools for automated test case generation, and can detect serious deeply embedded bugs.

6.1 GNU Coreutils

Experiment Setup. GNU Coreutils is a package of utilities widely used in Unix-like systems. The 87 programs from Coreutils (version 6.10) contain 20,559 lines of code, 988 functions, 14,450 branches according to `lcov` [40]. The program size ranges from 18 to 1,475 in lines, from 2 to 120 in functions, and from 6 to 1,272 in branches. It is an often-used benchmark for evaluating automated program analysis systems, including KLEE, MergePoint and others [5,12,41]. This is why we chose it as the benchmark to compare with KLEE and ANGR.

CRETE and ANGR generates test cases from program binaries without debug information, while KLEE requires program source code. To measure and compare the effectiveness of test cases generated from different systems, we rerun those tests on the binaries compiled with coverage flag and calculate the code coverage with `lcov`. Note that we only calculate the coverage of the code in GNU Coreutils itself, and do not compute code coverage of the library code.

We adopted the configuration parameters for those programs from KLEE's experiment instructions[1]. As specified in the instructions, we ran KLEE on each program for one hour with a memory limit of 1 GB. We increased the memory limit to 8 GB for the experiment on ANGR, while using the same timeout of one hour. CRETE utilizes a different timeout strategy, which is defined by *no new instructions being covered in a given time-bound*. We set the timeout for CRETE as 15 min in this experiment. This timeout strategy was also used by DASE [41] for its evaluation on Coreutils. We conduct our experiments on an Intel Core i7-3770 3.40 GHz CPU desktop with 16 GB memory running 64-bit Ubuntu 14.04.5. We built KLEE from its release v1.3.0 with LLVM 3.4, which was released on November 30, 2016. We built ANGR from its mainstream on Github at revision `e7df250`, which was committed on October 11, 2017. CRETE uses Ubuntu 12.04.5 as the guest OS for its VM front-end in our experiments.

Table 1. Comparison of overall and median coverage by KLEE, ANGR, and CRETE on Coreutils.

Cov.	Line (%)			Function (%)			Branch (%)		
	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
Overall	70.48	66.79	74.32	78.54	79.05	83.00	58.23	54.26	63.18
Median	88.09	81.62	86.60	100	100	100	79.31	70.59	77.57

Comparison with KLEE and ANGR. As shown in Table 1, our experiments demonstrate that CRETE achieves comparable test coverage to KLEE and generally outperforms ANGR. The major advantage of KLEE over CRETE is that it works on source code with all semantics information available. When the program size is small, symbolic execution is capable of exploring all feasible paths

[1] http://klee.github.io/docs/coreutils-experiments/.

Table 2. Distribution comparison of coverage achieved by KLEE, ANGR, and CRETE on COREUTILS.

Cov.	Line			Function			Branch		
	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
90–100%	40	24	33	65	60	65	15	16	19
80–90%	15	22	25	12	8	10	27	12	17
70–80%	13	14	10	3	7	5	14	16	25
60–70%	9	12	10	2	4	3	9	15	6
50–60%	5	7	4	1	4	1	8	11	9
40–50%	1	2	3	1	1	2	8	7	6
0–40%	4	6	2	3	3	1	6	10	5

with given resources, such as time and memory. This is why KLEE can achieve great code coverage, such as line coverage over 90%, on more programs than CRETE, as shown in Table 2. KLEE requires to maintain execution states for all paths being explored at once. This limitation becomes bigger when size of program gets bigger. What's more, KLEE analyzes programs within its own virtual environment with simplified model of real execution environment. Those models sometimes offer advantages to KLEE by reducing the complexity of the TBP, while sometimes they lead to disadvantages by introducing inaccurate environment. This is why CRETE gradually caught up in general as shown in Table 2. Specifically, CRETE gets higher line coverage on 33 programs, lower on 31 programs, and the same on other 23 programs. Figure 4(a) shows the coverage differences of CRETE over KLEE on all 87 COREUTILS programs. Note that our coverage results for KLEE are different from KLEE's paper. As discussed and reported in previous works [12,41], the coverage differences are mainly due to the major code changes of KLEE, an architecture change from 32-bit to 64-bit, and whether manual system call failures are introduced.

ANGR shares the same limitation as KLEE requiring to maintain multiple states and provide models for execution environment, while it shares the disadvantage of CRETE in having no access to semantics information. Moreover, ANGR provides models of environment at machine level supporting various platforms, which is more challenging compared with KLEE's model. What's more, we found and reported several crashes of ANGR from this evaluation, which also affects the result of ANGR. This is why ANGR performs worse than both KLEE and CRETE in this experiment. Figure 4(b) shows the coverage differences of CRETE over ANGR on all 87 COREUTILS programs. While CRETE outperformed ANGR on majority of the programs, there is one program `printf` that ANGR achieved over 40% better line coverage than CRETE, as shown in the left most column in Fig. 4(b). We found the reason is `printf` uses many string routines from `libc` to parse inputs and ANGR provides effective models for those string routines. Similarly, KLEE works much better on `printf` than CRETE.

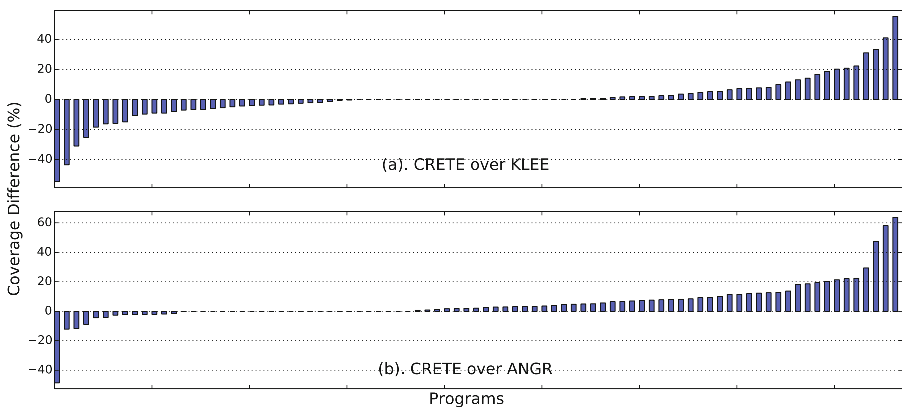

Fig. 4. Line coverage difference on COREUTILS by CRETE over KLEE and ANGR: positive values mean CRETE is better, and negative values mean CRETE is worse.

Coverage Improvement over Seed Test Case. Since CRETE is a concolic testing framework, it needs an initial seed test case to start the test of a TBP. The goal of this experiment is to show that CRETE can significantly increase the coverage achieved by the seed test case that the user provides. To demonstrate the effectiveness of CRETE, we set the non-file argument, the content of the input file and the `stdin` to zeros as the seed test case. Of course, well-crafted test cases from the users would be more meaningful and effective to serve as the initial test cases. Figure 5 shows the coverage improvement of each program. On average, the initial seed test case covers 17.61% of lines, 29.55% of functions, and 11.11% of branches. CRETE improves the line coverage by 56.71%, function coverage by 53.44%, and branch coverage by 52.14% respectively. The overall coverage improvement on all 87 COREUTILS programs is significant.

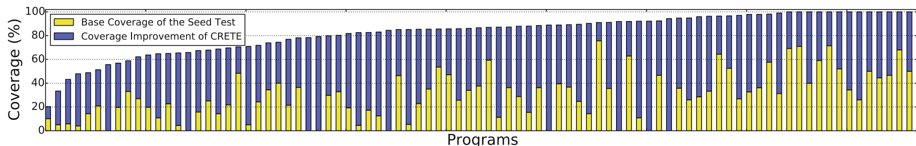

Fig. 5. Coverage improvement over seed test case by CRETE on GNU COREUTILS

Bug Detection. In our experiment on COREUTILS, CRETE was able to detect all three bugs on `mkdir`, `mkfifo`, and `mknod` that were detected by KLEE. This demonstrates that CRETE does not sacrifice bug detection capacity while working directly on binaries without debug and high-level semantic information.

6.2 TianoCore Utilities

Experiment Setup. TianoCore utility programs are part of the open-source project EDK2 [42], a cross-platform firmware development environment from Intel. It includes 16 command-line programs used to build BIOS images. The TianoCore utility programs we evaluated are from its mainstream on Github at revision `75ce7ef` committed on April 19, 2017. According to `lcov`, the 16 TianoCore utility programs contain 8,086 lines of code, 209 functions, and 4,404 branches. Note that we only calculate the coverage of the code for TianoCore utility programs themselves, and do not compute the coverage of libraries.

The configuration parameters we used on those utility programs are based on our rough high-level understanding of these programs from their user manuals. We assigned each program a long argument of 16 Bytes, and four short arguments of 2 Bytes, along with a file of 10 Kilobytes. We conduct our experiments on the same platform with the same host and guest OS as we did for the COREUTILS evaluation, and set the timeout also as 15 min for each program.

High Coverage Test Generation From Scratch. For all the arguments and file contents in the parameter configuration, we set their initial value as binary zeros to serve as the seed test case of CRETE. Figure 6 shows that CRETE delivered high code coverage, above 80% line coverage, on 9 out of 16 programs. On average, the initial seed test case covers 14.56% of lines, 28.71% of functions, and 12.38% of branches. CRETE improves the line coverage by 43.61%, function coverage by 41.63%, and branch coverage by 44.63% respectively. Some programs got lower coverage because of: (1) inadequate configuration parameters; (2) error handling code triggered only by failed system calls; (3) symbolic indices for arrays and files not well handled by CRETE.

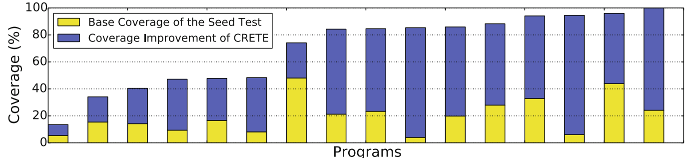

Fig. 6. Coverage improvement over seed test case by CRETE on TianoCore utilities

Bug Detection. To further demonstrate CRETE's capability in detecting deeply embedded bugs, we performed a set of evaluations focusing on concolic file with CRETE on TianoCore utility programs. From the build process of a tutorial image, OvmfPkg, from EDK2, we extracted 509 invocations to TianoCore utility programs and the corresponding intermediate files generated, among which 37 unique invocations cover 6 different programs. By taking parameter configurations from those 37 invocations and using their files as seed files, we ran CRETE with a timeout of 2 h on each setup, in which only files are made symbolic.

Table 3. Classified crashes found by CRETE on Tianocore utilities: 84 unique crashes from 8 programs

Crash type	Count	Severity	Crashed programs
Stack corruption	1	High (Exploitable)	VfrCompile
Heap error	6	High (Exploitable	GenFw
Write access violation	23	High (Exploitable)	EfiLdrImage, GenFw, EfiRom, GenFfs
Abort signal	2	Medium (Signs of exploitable)	GenFw
Read access violation	45	Low (May not exploitable)	GenSec, GenFw, Split, GenCrc32, VfrCompile
Other access violation	7	Mixed	GenFw

Combining experiments on concolic arguments and concolic files, CRETE found 84 distinct crashes (by stack hash) from eight TianoCore utility programs. We used a GDB extension [43] to classify the crashes, which is a popular way of classifying crashes for AFL users [44]. Table 3 shows that CRETE found various kinds of crashes including many exploitable ones, such as stack corruption, heap error, and write access violation. There are 8 crashes that are found with concolic arguments while the other 76 crashes are found with concolic files. We reported all those crashes to the TianoCore development team. So far, most of the crashes have been confirmed as real bugs, and ten of them have been fixed.

We now elaborate on a few sample crashes to demonstrate that the bugs found by CRETE are significant. `VfrCompile` crashed with a segmentation fault due to stack corruption when the input file name is malformed, e.g., `'\\.%*a'` as generated by CRETE. This bug is essentially a format string exploit. `VfrCompile` uses function `vsprintf()` to compose a new string from a format string and store it in a local array with a fixed size. When the format string is malicious, like `'%*a'`, function `vsprintf()` will keep reading from the stack and the local buffer will be overflowed, hence causing a stack corruption. Note that CRETE generated a well-formed prefix for the input, `'\\.'`, which is required to pass a preprocessing check from `VfrCompile`, so that the malicious format string can attack the vulnerable code.

CRETE also exposed several heap errors on `GenFw` by generating malformed input files. `GenFw` is used to generate a firmware image from an input file. The input file needs to follow a very precise file format, because `GenFw` checks the signature bytes to decide the input file type, uses complex nested structs to parse different sections of the file, and conducts many checks to ensure the input file is well-formed. Starting from a seed file of 223 Kilobyte extracted from EDK2's build process, CRETE automatically mutated 29 bytes in the file header. The mutated bytes introduced a particular combination of file signature and sizes and offsets of different sections of the file. This combination passed all checks on file format, and directed `GenFw` to a vulnerable function which mistakenly replaces the buffer already allocated for storing the input file with a much smaller buffer. Follow-up accesses of this malformed buffer caused overflow and heap corruption.

7 Conclusions and Future Work

In this paper, we have presented CRETE, a versatile binary-level concolic testing framework, which is designed to have an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution backends. At the core of this architecture is a standardized format for binary-level execution traces, which is LLVM-based, self-contained, and composable. Standardized execution traces are captured by concrete execution frontends, providing succinct and sufficient information for symbolic execution backends to reproduce the concrete executions. We have implemented CRETE with KLEE as the symbolic execution engine and multiple concrete execution frontends such as QEMU and 8051 Emulator. The evaluation of COREUTILS programs shows that CRETE achieved comparable code coverage as KLEE directly analyzing the source code of COREUTILS and generally outperformed ANGR. The evaluation of TianoCore utility programs found numerous exploitable bugs.

We are assembling a suite of 8051 binaries for evaluating CRETE and will report the results in the near future. Also as future work, we will develop new CRETE tracing plugins, e.g., for concrete execution on physical machines based on PIN. With these new plugins, we will focus on synthesizing abstract system-level traces from trace segments captured from binaries executing on various platforms. Another technical challenge that we plan to address is how to handle symbolic indices for arrays and files, so code coverage can be further improved.

Acknowledgment. This research received financial supports from National Science Foundation Grant #: CNS-1422067, Semiconductor Research Corporation Contract #: 2708.001, and gifts from Intel Corporation.

References

1. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**, 385–394 (1976)
2. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference (2005)
3. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005) (2005)
4. Song, D., et al.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7_1
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008) (2008)
6. Chipounov, V., Kuznetsov, V., Candea, G.: The s2e platform: design, implementation, and applications. ACM Trans. Comput. Syst. **30**, 1–49 (2012)
7. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Commun. ACM **10**, 1–20 (2012)

8. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (2012)
9. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**, 82–90 (2013)
10. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: PLDI 2012 (2012)
11. Marinescu, P.D., Cadar, C.: Make test-zesti: a symbolic execution solution for improving regression testing. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012) (2012)
12. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: ICSE 2014 (2014)
13. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**, 74–84 (2014)
14. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., et al.: SOK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
15. Stephens, N., Grosen, J., Salls, C., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (2016)
16. Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Bootstomp: on the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium (2017)
17. Palikareva, H., Kuchta, T., Cadar, C.: Shadow of a doubt: testing for divergences between software versions. In: ICSE 2016 (2016)
18. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_3
19. Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (2014)
20. Kasikci, B., Zamfir, C., Candea, G.: Automated classification of data races under both strong and weak memory models. ACM Trans. Program. Lang. Syst. **37**, 1–44 (2015)
21. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: correctness checking for real code. In: Proceedings of the 24th USENIX Conference on Security Symposium (2015)
22. Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., Xie, T.: Automated test input generation for android: towards getting there in an industrial case. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (2017)
23. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference (2005)
24. Kasolik, M.: 8051 emulator. http://emu51.sourceforge.net/
25. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (2004)
26. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI 2007 (2007)
27. Godefroid, P.: Random testing for security: blackbox vs. whitebox fuzzing. In: Proceedings of the 2nd International Workshop on Random Testing (2007)

28. AFL: American fuzzy lop. http://lcamtuf.coredump.cx/afl/
29. Luk, C.K., Cohn, R., Muth, R., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005 (2005)
30. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (2012)
31. Lattner, C., Adve, V.: LLVM language reference manual (2006). http://llvm.org/docs/LangRef.html
32. Lattner, C.: LLVM and Clang: next generation compiler technology. In: The BSD Conference (2008)
33. Dhurjati, D., Kowshik, S., Adve, V.: Safecode: enforcing alias analysis for weakly typed languages. In: PLDI 2006 (2006)
34. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: VMKit: a substrate for managed runtime environments. In: Proceedings of the 6th ACM International Conference on Virtual Execution Environments (2010)
35. Grosser, T., Größlinger, A., Lengauer, C.: Polly-performing polyhedral optimizations on a low-level intermediate representation. Parall. Process. Lett. **22**, 1–28 (2012)
36. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the IEEE Symposium on Security and Privacy (2010)
37. Chipounov, V., Candea, G.: Dynamically translating x86 to LLVM using QEMU. Technical report EPFL-TR-149975 (2010)
38. GNU: GNU coreutils - core utilities. https://www.gnu.org/s/coreutils
39. Tianocore: Tianocore. http://www.tianocore.org/
40. Oberparleiter, P.: A graphical front-end for gcc's coverage testing tool gcov. http://ltp.sourceforge.net/coverage/lcov.php
41. Wong, E., Zhang, L., Wang, S., Liu, T., Tan, L.: Dase: document-assisted symbolic execution for improving automated software testing. In: ICSE 2015 (2015)
42. Tianocore: EDK II. https://github.com/tianocore/edk2
43. Foote, J.: The 'exploitable' gdb plugin. https://github.com/jfoote/exploitable
44. AFL-Utils: Utilities for automated crash sample processing/analysis. https://github.com/rc0r/afl-utils

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.