FirVer: Concolic Testing for Systematic Validation of Firmware Binaries

Tashfia Alam[1], Zhenkun Yang[2], Bo Chen[2], Nicholas Armour[2] and Sandip Ray[1]

[1]ECE Department, University of Florida, Gainesville, FL, USA
[2]Intel Corporation, Hillsboro, OR, USA

Abstract— We present an infrastructure, FirVer, for systematic validation of firmware binaries. FirVer makes unique use of virtual prototyping and unit testing interfaces for effective comprehension of hardware-firmware. We used FirVer on several library functions of TianoCore, a full-featured UEFI-compatible boot firmware developed by Intel Corporation. FirVer achieved more than 90% in line and function coverages, and between 60% and 80% branch coverage. FirVer also enabled exploration of corner cases that exposed segmentation faults in many constituent functions.

I. Introduction

Firmware is software that can directly control hardware components and is shipped with the hardware platform [1]. The hardware-specific, low-level nature of firmware distinguishes it from application code or even the operating system (OS), which for the most part, tend to be device-independent. In some cases, firmware provides an interface to the rest of the OS; in other instances, firmware is executed during the computer's boot process, *e.g.*, the Basic Input/Output System (BIOS) executes before the OS is loaded. In virtually every case, firmware executes in privileged mode. Consequently, the correct functionality of firmware is critical to the trustworthiness of the overall system. Furthermore, it is depressingly easy to exploit firmware vulnerabilities to compromise security [2], [3]. Obviously, validation techniques to enable systematic detection of firmware errors are of paramount importance to system assurance.

Firmware validation has received considerable attention over the recent years [4], [5], [6], [7]. Much of this work aims to adapt software verification approaches to firmware, possibly accounting for sparing and well-defined interfaces with hardware. Unfortunately, firmware poses some unique challenges that make such adaptation challenging. In particular, given the tight interaction of firmware with hardware, firmware validation is really a hardware-software co-validation problem; however, methodologies, flows, formalisms used in practice differ significantly for hardware and software validation flows.

In this paper, we develop an infrastructure for firmware validation that addresses the above constraints. Our infrastructure, FirVer, (for "Firmware Verification Infrastructure") enables automated, binary-level concolic testing for systematic exploration of firmware implementations. We account for hardware-software interactions by analyzing the binary on top of a virtual platform that includes interface definitions for the target hardware (*e.g.*, registers, interface protocols) visible to the firmware. A unique contribution of FirVer is a methodology for concolic testing directly on top of a virtual machine without requiring a guest OS to facilitate trace recording and analysis. Finally, it avoids the need for supporting specific source language features by focusing on firmware exploration at the binary level.

We have applied FirVer to test of TianoCore (http://www.tianocore.org), a complex, full-featured boot firmware developed by Intel Corporation. TianoCore is an open-source implementation of the Unified Extensible Firmware Interface (UEFI) developed as the open-source replacement for legacy BIOS in personal computers. Our experimental results demonstrate that the method provides high assurance in firmware validation, *e.g.*, for the tested libraries, FirVer achieved line and function coverage above 90%. *Perhaps more importantly,* FirVer *could systematically expose several segmentation fault errors in the current TianoCore implementation.*

The remainder of the paper is organized as follows. Section II provides the requisite background on concolic testing and a brief overview of binary-level concolic testing. We discuss the challenges in applying existing concolic testing frameworks directly on firmware binaries and present a solution, FirVer, to extend it on firmware in section III. Section IV illustrates the FirVer design components and we show the application and evaluation of FirVer on TianoCore functions in the following section. We discuss some related work in Section VI and conclude in Section VII.

II. Background

A. Symbolic Execution and Concolic Testing

The idea of symbolic execution is to treat certain variables as having symbolic values. The symbolic execution creates a symbolic expression for each variable involved (in the program under test) as follows. Consider the code fragment below.

```
void test (int x, int y) {
    int z = 2*y;
    int p,q;
    if (x ==100) {
        if (x < z) {
            p =2q;
    }}}
```

The symbolic expression for `p` after symbolic execution of the code fragment will be the following:

```
if (x == 100) then
   {if (x < 2*y0) then p = 2*q0 else q0}
```

Since the symbolic expression represents the entire set of values the symbolic variable can take, symbolic execution can point to subtle corner case scenarios and hard-to-excite

bugs, *e.g.*, scenarios corresponding to dereferencing an invalid pointer or reading past the end of a buffer.

The name "concolic" is a portmanteau of "concrete" and "symbolic". Concolic testing combines concrete simulations with symbolic execution. The software under test is first executed on concrete inputs, and the execution trace is recorded, which includes information of the branches taken along with any other pertinent context. The execution trace is then fed to a symbolic execution engine (SEE); the SEE treats some (or all) of the inputs from concrete tests as symbolic and constructs the expressions involving these symbolic values that are built up as it executes the sequence of operations in the trace. The goal is to select another execution path, *e.g.*, by flipping some of the conditionals in the current concrete path. This is done by the symbolic execution engine. The branches in the trace with conditionals involving symbolic values are branches whose outcomes could be changed with an appropriate choice of concrete values for the symbolic values. For instance, suppose concolic testing starts with the inputs `x = 0, y = 0`. Here, the branch being encountered is the test `(x != 100)`; consequently, the variable `x` is a candidate variable whose value can be flipped to explore a different branch. A constraint solver can find these concrete values, *e.g.*, in this case, the assignment is `x != 100` and produce a new test case exercising a new path through the program. These new test cases are then, in turn, concretely executed. The process repeats until all possible paths through the software have been explored or a user-specified constraint is satisfied.

B. Binary Concolic Testing

Enabling symbolic execution for binary code requires keeping track of the program state during a concrete run. To achieve this, binary concolic testing tools are generally built on top of a virtual platform (VP) environment. The idea is to run the target binary on top of a guest OS executed on the VP; the guest OS can be instrumented to provide a controlled execution environment for the binary and record the execution environment, which is used by the SEE. This provides a context to symbolic execution and is instrumental to the generation of effective tests. A popular VP infrastructure used in concolic testing tools is the Quick Emulator (QEMU) virtual prototyping environment (https://qemu-project.gitlab.io/qemu). It includes support for X86 ISA, and consquently can run X86 binaries of Linux and Windows. Correspondingly, KLEE [8] is a commonly used SEE used for software analysis.

Concolic testing frameworks built with QEMU and KLEE include S^2E [9] and CRETE [10]. Both use trace recording with guest OS on QEMU VP, which is used by KLEE to generate new tests. Our work builds upon the CRETE framework, which enables loose coupling of concrete and symbolic executions through standardized execution traces and test cases. The key idea is that one can execute the binary on top of a virtual prototyping (VP) environment and record the concrete execution trace from the VP together with sufficient context to enable offline symbolic execution.

III. Firmware Analysis: Challenges and the FirVer Solution

Although symbolic exploration and concolic testing have been successfully applied on application and system code, its application on firmware has been limited. Source-level symbolic exploration is complicated by the need to include models of hardware to comprehend hardware-firmware interactions that are realistic enough to exhibit the interaction corner cases while abstract enough to eliminate irrelevant details; building and maintaining such models for each interaction scenario is non-trivial and labor-intensive. Furthermore, firmware source code (when available) includes complex low-level language features (*e.g.*, it is common to have several levels of redirections of function pointers and a significant amount of embedded assembly [11]), making it infeasible to apply verification techniques that require neat language features and formalisms. On the other hand, binary-level concolic testing is limited in practice by the need for instrumenting and recording binary traces, as explained above, and generally exploit the services of a guest OS running on a VP. Unfortunately, firmware, by its very nature, runs on "bare metal", *i.e.*, on top of the hardware directly, where OS services are unavailable.

The key insight for FirVer is that it is possible to overcome the challenges in binary-level concolic testing without requiring the addition of instrumentation or trace recording functionalities implemented through operating systems by directly using available VP features. Note that capturing the context of concrete execution is necessary for concolic testing to enable dynamic sequencing of concrete and symbolic executions. Instead, FirVer simply records (in VP) the sequence (trace) $\mathcal{T}$ of instructions encountered during a concrete firmware run. Offline analysis of $\mathcal{T}$ can then identify (1) the most recent conditional encountered and (2) test inputs required to flip the condition to generate a new run. The approach significantly decouples the concrete and symbolic executions by enabling symbolic execution offline while still exploiting the benefit of using symbolic analysis to iteratively identify test inputs for systematic exploration of firmware paths. Note that this approach can be integrated directly into a VP without requiring OS support; transfer of relevant information can be performed directly from the VP to the offline SEE through simple hypercalls.

IV. FirVer Design

FirVer implements the observations above while abstracting the trace recording and transferring hypercalls from the user. Algorithm 1 shows the overall FirVer execution flow. To provide this transparency, the front end of FirVer is simply a test harness (referred to as FirVer *runner*) that enables the user to specify the target firmware. In addition to the regular test harness used in unit testing, the FirVer harness enables the stipulation of specific variables to be concolic. Additionally, The trace hypercall passes this information to the SEE (see below) to enable focused symbolic exploration. The FirVer infrastructure extends a VP framework with the following components.

Runner executes a concrete run of the firmware, using a test input provided by the *Replayer* (see below). The runner

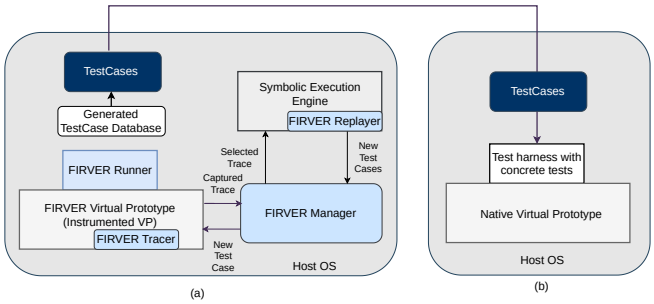

Fig. 1. Realization of the FirVer Solution on Firmware. (a) FirVer Architecture during Test Generation. (b) Test Execution.

execution is seeded by an initial (default) test input provided by the user.

Tracer captures the binary trace $\mathcal{T}$ from the concrete execution, together with the hardware state C reached at the end of the concrete run. The tracer coordinates with the *Trace Manager* (see below) through VP hypercalls.

Manager coordinates between the concrete and symbolic environments. Algorithm 2 summarizes the *Manager* functionality. It includes a database of trace pairs $\langle \mathcal{T}, C \rangle$ obtained from the *Tracer*. Since $\mathcal{T}$ includes the sequence of instructions encountered during concrete execution, we can apply simple control flow analysis to identify the basic blocks in $\mathcal{T}$. Furthermore, hardware interaction is computed by correlating the encountered instructions in $\mathcal{T}$ with the hardware states captured in C, *e.g.*, update to memory location $\mathcal{L}$ in C would be explicitly captured by a store instruction in $\mathcal{T}$.[1] The "massaged" trace $\mathcal{T}'$ is added to the Trace-database, $\mathcal{V}$ and one of the selected trace is passed into the Replayer (see below); correspondingly, the generated tests, $\mathcal{B}$ from Replayer are added to the Test database, $\mathcal{U}$ and one (a set) of the previously unexplored tests, t are passed to the runner/concrete execution environment.

Replayer. The Replayer is embedded in the symbolic execution environment. Its job is to invoke SEE and coordinate with the Manager to receive traces and generate tests. SEE is simply an off-the-shelf symbolic exploration tool that computes inputs to explore a feasible unexplored branch consistent with the hardware state C, which can be solved as a Boolean Satisfiability problem. Unlike traditional concolic testing techniques, we only compute an unexplored control flow from the given trace rather than exploring the entire firmware state space. Consequently, the SEE incurs little cost in performance or memory.

Remark 1: One challenging factor in designing the runner is the identification of the firmware variables permitted to be symbolic. For instance, consider a variable `v` that holds the pointer to a buffer. Obviously, making the pointer value (which would mean that starting address of the buffer) symbolic is not helpful. FirVer runner accounts for such cases, *e.g.*, instead of the pointer, the buffer entries pointed by the variable (and the size of the buffer) are made symbolic in the test harness (and correspondingly the runner). Handling such details is critical in making concolic testing tool viable for complex low-level software and firmware implementations that include realistic features.

[1]Since $\mathcal{T}$ is a concrete run in VP, all memory indirections are obviously resolved prior to the execution of the instruction. Consequently, it is possible to perform the correlation using only the pair $\langle \mathcal{T}, C \rangle$.

Algorithm 1: FIRVER Test Generation

Input:
(a) Code Under test ($\mathcal{TP}$)
(b) Initial concrete seed test τ
Output: Database of Tests $\mathcal{U}$

$\mathcal{U} \leftarrow \{\tau\}$
$\mathcal{V} \leftarrow \{\}$
repeat
  Let $t \in \mathcal{U}$ be a previously unexplored test
  $\langle \mathcal{E} \rangle \leftarrow$ `Runner(`$\mathcal{TP}$`,t)`
  $\langle \mathcal{T}, C \rangle \leftarrow$ `Tracer(`$\mathcal{E}$`)`
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{\langle \mathcal{T}, C \rangle\}$
  $\langle \mathcal{U}, \mathcal{V} \rangle \leftarrow$ Manager$(\mathcal{U}, \mathcal{V})$
until *No New trace added to* $\mathcal{V}$
return $\mathcal{U}$

Algorithm 2: Managing traces and tests

Input:
(a) Database of Tests $\mathcal{U}$
(b) Database of Trace Pairs $\mathcal{V}$
Output: New test and trace databases $\langle \mathcal{U}, \mathcal{V} \rangle$

Let $\langle \mathcal{T}, C \rangle \in \mathcal{V}$
Let $\mathcal{V} \leftarrow \mathcal{V} \backslash \langle \mathcal{T}, C \rangle$
$\mathcal{T}' \leftarrow$ MassageTrace$(\mathcal{T}, C)$
$\langle x, B \rangle \leftarrow$ SEE$(\mathcal{T}')$
$\mathcal{V} \leftarrow \mathcal{V} \cup \{x\}$
$\mathcal{U} \leftarrow \mathcal{U} \cup B$
return $\langle \mathcal{U}, \mathcal{V} \rangle$

Fig. 1 shows the system architecture of FirVer. Note that the FirVer runner is created by simply augmenting the standard test harnesses used for unit testing, with facilities for capturing additional book-keeping information required by Tracer. Correspondingly, the tests generated by FirVer can be executed systematically on top of the same virtual prototype environment (Native VP without any Tracer program) by simply replacing the FirVer runner with a traditional test harness.

V. Application: Application and Evaluation of FirVer on TianoCore Boot Firmware

A. Setup

We have used FirVer to verify several functions of the TianoCore firmware. TianoCore was developed by Intel Corporation as the "Foundation Code" of its Extensible Firmware Interface (EFI), a successor to the 16-bit x86 legacy PC BIOS [12]. TianoCore is a full-featured boot firmware, including all five phases of UEFI. Additionally, it has a full firmware stack for power management, security, update signing, glitch resistance, and many others.

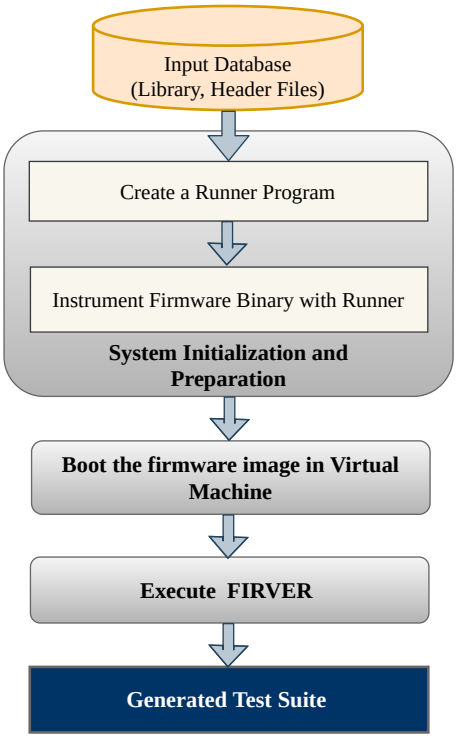

Fig. 2. FirVer workflow applied on TianoCore

Our realization of FirVer uses QEMU-x86 as the VP, with KLEE as the target SEE. Fig. 2 shows the FirVer workflow for TianoCore. The FirVer setup requires a test harness definition that is used to create the runner program. Note that since TianoCore is UEFI-compatible, the test harnesses can be built as an EFI application.

Remark 2: (Implementation Note) The UEFI application can be mounted in QEMU using the VFAT file system. However, for QEMU to create a snapshot, it has to save the metadata related to the snapshot into the actual images used to boot. It means the image(s) we use to boot (for example, `bios.bin` in FAT32 file-system) need to be in the proper image format compatible with QEMU. We address this by creating a blank `iso` image with a vfat file-system that contains the harness app and convert the `iso` image into QEMU `qcow2` format.

We applied FirVer on two TianoCore libraries BaseSafeIntLib and MemoryAllocationLib. BaseSafeIntLib library contains implementation of "safe" operation of type conversion, addition, subtraction and multiplication functions of different sizes (bytes) of data with the goal to eliminate integer overflows. MemoryAllocationLib is responsible for safe memory allocation routines depending on boot services of DXE phase of UEFI execution. These functions constitute the regression suite of HBFA (Host Based Firmware Analyzer) unit testing framework on BIOS [13] and are critical to the security of the firmware. Table I provides a summary of some of the functions. ***All functions from the two libraries (***90 ***from*** **MemoryAllocationLib** ***and*** 27 ***from*** **MemoryAllocationLib***) were explored in our experiments.***

Remark 3: In addition to the function itself, the number of test cases generated depends on the allocated buffer size in the test harness (if the function involves usage of buffer). Indeed, in many cases, a minimum size is necessary for appropriate test cases to be generated, *e.g.*, for `SafeUint64Mult`, the appropriate buffer size is required for the concolic testing to generate tests that correspond to valid multiplicands and multiplier.

B. Coverage Results

Table II shows the evaluation of generated tests in terms of line coverage, function coverage, and branch coverage of the functions from the mentioned libraries. For a succinct presentation of the results, we divide the functions into a few classes based on the functionality and report the respective coverage for each class. For example, the SafeUintMult class represents the functions that prevent overflow during the multiplication of unsigned integers for different sizes of multiplicands and multipliers. AllocatePages includes functions that allocate one or more 4 KB pages of different data types, such as EfiBootServicesData, EfiRuntimeServicesData. The line coverage from FirVer is 90-100%, and the function coverage is between 87-100%.

It is illustrative to understand the branch coverage results, which are much lower than other coverages. There are two reasons for this behavior. First, certain branches are semantically impossible to cover. Consider, for example, the following code fragment.

```
if (x < 100) then return;
...
```

Clearly, the code represented as `...` (including any branch conditions there) is only reachable when the value of `x` is not less than `100`. However, branch coverage will attempt to reach those control flow under both conditions `(x < 100)` and `(x ≥ 100)` resulting in the coverage values being penalized. Our "estimated average" column adjusts for this approximation by eliminating the semantically unreachable branch conditions. The second reason is that there were some corner cases where exploring a branch condition led to segmentation faults (see below). Consequently, we had to explicitly remove those tests from coverage calculation.

C. Segmentation Faults

The FirVer test generation found segmentation fault under a specific corner case bug that manifested in 63 functions of BaseSafeIntLib. This happens if a test input has a `NULL` value for the buffer passed to the function. The buffer is used by these functions to store the output (converted) value, avoiding explicit returns. Consequently, if enough space is not allocated for the buffer, the output generated goes out of bound, resulting in a segmentation fault. Note that while this can be considered a real bug, we have not found an instance of the *invocation* of that function that resulted in this corner case being exposed. Nevertheless, the fact that FirVer found segmentation faults in a firmware implementation that is defined as a foundation code for next-generation boot firmware underlines the importance of tools like FirVer for firmware analysis.

TABLE I
FIRVER APPLIED ON TIANOCORE

Target functions	Significance of the functions	Allocated Buffer Size in byte in the test harness	Number of Test cases
SafeInt8ToUint16	It performs 8 bit unsigned integer to 16 bit unsigned integer conversion. This function converts the input specified by the output type and stores the result into allocated output buffer. So the output must be at least as large as the specified output type. If overflow or underflow occurs, it returns error.	2	5
SafeInt16Add	This function performs addition on 16 bit integers. It performs addition using provided augend and addend and returns the output stored in a buffer. The buffer should be st least the size of the output type, otherwise will return an error in case of an overflow or underflow.	2	10
SafeUint64Mult	It performs multiplication on uint64 type. It takes the multiplicand and multiplier as input and converts the result specified by the output type and stores the output in a buffer. The buffer is at least as large as the result type.If overflow/underflow condition occurs,it returns error.	8	177
SafeInt64Sub	It performs subtraction on unsigned 64 bits integers. It performs subtraction using the provided minuend and subtrahend and return the output stored in a buffer in the requested format. The buffer should be at least the size of the output type, returns an error in case of overflow/underflow.	8	94
AllocatePages	It allocates one or more 4KB pages of EfiBootServicesData type and returns a pointer to the allocated buffer. In case of 0 page or insufficient memory satisfying the service, NULL is returned.	same as the input size	2
AllocateCopyPool	It allocates space or the number bytes specified by allocation size of type EfiBootServicesData, copies allocation size bytes from buffer to the newly allocated buffer, and returns a pointer to the allocated buffer. In case of zero allocation size or insufficient memory satisfying the service, NULL is returned.	same as the input size	thousands of tests

VI. RELATED WORK

DART [14] is one of the early works in concolic testing. It applies automated random testing on the source code. KLEE [8] is a highly used symbolic execution tool at the source level which automatically generates tests with high coverage. It performs source-level symbolic execution while our work FIRVER uses KLEE as the symbolic executor and extends it to perform concolic execution on firmware binaries. UC-KLEE [15] is an extension of KLEE that performs under-constrained symbolic execution. FIE [16] also uses KLEE as SEE to apply on low power platforms, especially firmware programs for the MSP430 family of micro-controllers. S^2E [9] and CRETE [10], [17] tools have been developed to perform concolic testing on binaries. However, their applicability is limited to application code or system code under guest OS support. Another symbolic execution-based tool is S2E [9] which provides automated path analysis that tightly couples the concrete and symbolic execution. Our work with FIRVER decouples concrete execution from symbolic analysis and provides the flexibility of replacing the infrastructure used on the VP component with a test harness while leaving the offline analysis essentially unchanged. Another related line of research is by Ahn *et al.* [18] that virtualizes the hardware/firmware interactions as transaction properties and applies concolic testing. However, it is not applicable when the transaction does not exhibit a producer-consumer relationship.

One of the early tools in the security analysis of binary code is BitBlaze [19]. It builds a binary analysis platform and uses it to provide solutions for different security vulnerabilities. ANGR [20] is a python based framework for binary analysis. However, it suffers memory overhead as it has to maintain execution states for all paths while FIRVER minimizes memory usage by analyzing the system path by path. Vex [21] integrates some of the existing binary analysis approaches based on both static and dynamic techniques and reproduces a single framework. Aside from concolic testing, fuzzing has also been successfully used for the detection of bugs or security loopholes in low-level software [22]. AFL [23] is a popular fuzzer that incorporates instrumentation and genetic algorithm to generate test cases. JFuzz [24] is a tool that applies concolic testing on Java programs, and Pex [22] applies testing on .NET programs. Fuzzing perturbs inputs randomly to provide unexpected inputs to the software/firmware under test. The key difference with concolic testing is that the latter systematically targets different program branches through symbolic analysis of the branches encountered. Obviously, when successful, concolic testing provides a more systematic means for exploring corner cases of the target design. HBFA [25] uses fuzzing for UEFI. However, it confines analysis to the software functionality and abstracts hardware interactions.

TABLE II
EVALUATION OF THE GENERATED TESTS APPLIED ON TIANOCORE

Function Class	Number of generated tests		Line Coverage (%)			Function Coverage (%)			Branch Coverage (%)			
	Min	Max	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Estimated Avg
SafeUintMult	6	177	94	100	95.2	75	100	90.625	57.7	88.9	73.3	83.7
SafeIntMult	7	198	91	95.8	94	75	100	92.85	70.6	83.3	76.92	81.35
SafeUintAdd	5	46	90.9	100	96.36	100	100	100	57	62.16	59.75	95
SafeIntAdd	6	46	95.8	100	97.26	75	100	91.66	75	100	81.25	87.5
SafeUintSub	5	188	91.3	91.3	91.3	100	100	100	62.5	62.5	62.5	85.5
SafeIntSub	7	213	91.7	95.2	93.4	100	100	100	70.2	83.3	73.8	87.5
SafeUintToInt	5	35	75	82	78.5	75	100	87.5	57.7	57.7	57.7	67.86
SafeIntToUint	6	53	76.5	81	79.2	66.7	100	83.35	50	50	50	65.013
AllocatePages	2	26	100	100	100	100	100	100	66.7	66.7	66.7	100
AllocatePool	27	29	100	100	100	100	100	100	66.7	66.7	66.7	100
Reallocatepool	139	198	100	100	100	100	100	100	66.7	66.7	66.7	100
AllocateCopyPool	-	thousands in an hour	100	100	100	100	100	100	66.7	66.7	66.7	100

VII. CONCLUSION AND FUTURE WORK

Firmware validation is a crucial component in developing high-assurance SoC designs and cannot be directly addressed by traditional software validation techniques. We developed a concolic testing framework extended on a virtual prototyping environment for systematically validating firmware binaries. We demonstrated the effectiveness of FIRVER libraries of TianoCore boot firmware. In addition to achieving high coverage, the tests generated illustrated corner case scenarios that result in segmentation faults.

In future work, we plan to apply the FIRVER system more extensively on BIOS and also explore its applicability on other (microcontroller) firmware. One target is the power management firmware in SoC designs. We will also explore opportunities for more automation, *e.g.*, in generating the test harness, smart identification of symbolic parameters.

REFERENCES

[1] J. Grundy, "Firmware Verification: Challenges and Progress," in *FM-CAD*, 2013.

[2] C. Kallenberg and X. Kovah, "How Many Million BIOSes Would you Like to Infect?," in *CanSecWest*, 2015.

[3] J. Loucaides and A. Furtak, "A new class of vulnerability in SMI Handlers of BIOS/UEFI Firmware," in *CanSecWest*, 2015.

[4] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, "Verifying information flow properties of firmware using symbolic execution," in *DATE*, pp. 337–342, 2016.

[5] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware," in *DAC*, pp. 91:1–91:6, 2018.

[6] D. Kroening, L. Liang, T. Melham, P. Schrammel, and M. Tautschnig, "Effective verification of low-level software with nested interrupts," in *DATE*, pp. 229–234, 2015.

[7] S. Ahn, S. Malik, and A. Gupta, "Completeness bounds and sequentialization for model checking of interacting firmware and hardware," in *CODES+ISSS*, pp. 202–211, 2015.

[8] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, 2008.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

[10] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering* (A. Russo and A. Schürr, eds.), (Cham), pp. 281–298, Springer International Publishing, 2018.

[11] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer, "Excite: Symbolic execution for BIOS security," in *USENIX Workshop on Offensive Technologies*, 2015.

[12] "Unified Extensible Firmware Interface." http://www.uefi.org.

[13] "HBFA," 2021. https://github.com/tianocore/edk2-staging/tree/HBFA.

[14] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *PLDI*, p. 213–223, 2005.

[15] D. Ramos, D. Engler, A. Aiken, D. Dill, and S. U. C. S. Department, *Under-constrained Symbolic Execution: Correctness Checking for Real Code*. Stanford University, 2015.

[16] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security Symposium*, 2013.

[17] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pp. 1–8, IEEE, 2019.

[18] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[19] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information Systems Security* (R. Sekar and A. K. Pujari, eds.), (Berlin, Heidelberg), pp. 1–25, Springer Berlin Heidelberg, 2008.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, 2016.

[21] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[22] N. Tillmann and J. de Halleux, "Pex–white box test generation for .net," in *Tests and Proofs* (B. Beckert and R. Hähnle, eds.), (Berlin, Heidelberg), pp. 134–153, Springer Berlin Heidelberg, 2008.

[23] "AFL." https://lcamtuf.coredump.cx/afl/.

[24] H. Zhu, "Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods," in *2015 Second International Conference on Trustworthy Systems and Their Applications*, pp. 8–15, 2015.

[25] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer, "Uefi firmware fuzzing with simics virtual platform," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.