Symbolic execution for BIOS security[1]

Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R. Tuttle, Vincent Zimmer[2]
Intel Corporation

May 25, 2015

Abstract

We are building a tool that uses symbolic execution to search for BIOS security vulnerabilities including dangerous memory references (call outs) by SMM interrupt handlers in UEFI-compliant implementations of BIOS. Our tool currently applies only to interrupt handlers for SMM variables. Given a snapshot of SMRAM, the base address of SMRAM, and the address of the variable interrupt handler in SMRAM, the tool uses S^2E to run the KLEE symbolic execution engine to search for concrete examples of a call to the interrupt handler that causes the handler to read memory outside of SMRAM. This is a work in progress. We discuss our approach, our current status, our plans for the tool, and the obstacles we face.

1 BIOS security

BIOS security — a hot topic for years among technical professionals with nearly a decade of high-quality work finding and protecting against security vulnerabilities — finally hit the mainstream press with a bang when *Forbes* magazine reported [1] on results presented at the CanSecWest conference in March 2015. At that conference alone, four talks [2] [3] [4] [5] discussed security vulnerabilities related to the System Management Mode on the Intel processor architecture running BIOS implementing UEFI, the Unified Extensible Firmware Interface [6], an industry-standard specification and architecture for BIOS.

System Management Mode (SMM) is the highest, most privileged state of execution on Intel-based platforms. Code running in system management mode has complete, unfettered access to everything, including all registers and all memory and all devices attached to the processor. Code running in system management mode lives in a small, protected region of memory (SMRAM) made invisible to everything but code running in system management mode. It is the perfect place to hide a root kit: it is invisible to tools like virus checking software that might otherwise be able to find it, and code within it has complete access to the machine, able to read and write everything. Unfortunately, clever teams have demonstrated for years that this is distressingly easy to do.

This point exploded into public view [1] at the CanSecWest conference in March 2015. Among several interesting results was a paper provocatively titled "How many million BIOSes would you like to infect?" [2]. The authors made the following observation: Almost all machines are vulnerable because almost all machines are running unpatched BIOS (most consumers don't know patches exist, and even sophisticated consumers apply patches with their fingers crossed), and widespread software reuse in the BIOS community (normally considered an enlightened approach to software development) means that almost all machines are vulnerable to the same attacks. As a result, attacks can be automated in a reliable fashion. The authors drove home this point by demonstrating that in a database of over 3000 BIOS images from shipping platforms, each one was vulnerable to at least one of three known vulnerabilities in SMM, and only seven were not vulnerable to all three.

Code running in system management mode (SMM) is intended to be trusted code sitting in a protected region of memory (SMRAM) and accessing only trusted components of the system. It would be a severe security vulnerability if trusted code in SMRAM could be tricked into executing untrusted code outside of SMRAM, or into reading or writing data outside of SMRAM under the control of an attacker. It would. And it is, as recent work has demonstrated so compellingly.

Hardware mitigation is available. On recent Intel processors, if a bit is set in a machine status register, a machine check is generated if code running in SMM tries access memory outside of SMRAM. Too often, however, systems ship with this security feature turned off. Because BIOS is written so late in product development, and because evidence of the machine bricking due to SMM-related machine checks is discovered so close to product ship dates, the pressure to simply turn off this security feature and ship can be overwhelming. (Software mitigation --- such as making pages outside SMRAM un-executable in SMM --- suffer a similar fate).

[1] This paper appeared at the Ninth USENIX Workshop on Offensive Technologies (WOOT '15) on August 10–11, 2015.
[2] The authors can be reached at oleksandr.bazhaniuk@intel.com, john.loucaides@intel.com, lee.g.rosenbaum@intel.com, mark.r.tuttle@intel.com, vincent.zimmer@intel.com.

Our goal is to find these vulnerabilities (and the inputs inducing them) in the earliest stages of product development, and give developers a chance to remove them.

2 BIOS validation

BIOS validation of SMM is accomplished today in an *ad hoc* fashion. BIOS health may be judged using criteria as basic as "Does the operating system boot?" and "Do the test suites for the operating system pass?" Unfortunately, SMM correctness may not be visible or testable at the level of the operating system. The next few paragraphs assume some familiarity with the UEFI specification [7] and implementation architecture [8]. The bottom line, however, is that validation and debugging of SMM can be as primitive as setting breakpoints and stepping through code on a development board with the debug equivalent of printf statements, and we want to do better.

There are heuristics employed to detect errant behavior of SMM handlers. For example, on 64-bit machines, the UEFI PI SMM [6] software model requires that the PI SMM handlers run in the same mode as the DXE, namely 64-bit long mode with paging. Implementations often perform one-to-one virtual-to-physical mapping of SMRAM, with page faults for accesses outside of SMRAM. On a debug build, the page fault handler can log the accesses. On a release build, though, there is no such observability. Debugging UEFI SMM drivers is a difficult and low-level task for BIOS developers since SMM code cannot be viewed when SMRAM is closed and locked. Specifically, leaving SMRAM accessible after installation of the EFI_SMM_READY_TO_LOCK_PROTOCOL from the PI specification poses a potential privilege escalation since third-party code should not have the opportunity to install itself. The above protocol is invoked prior to the invocation of third-party UEFI drivers and applications.

A hardware-based JTAG debugger, such as the Intel® In Target Probe (ITP) [9], provides the ability to break on SMM entry, allowing engineers to step through SMM code at the hardware level. The UEFI Development Kit (UDK) at www.tianocore.org provides a SourceLevelDebugPkg containing a SmmDebugAgent with some SMM debug capabilities in a Windows WinDbg or Linux GDB environment. The UDK also provides a DEBUG() macro that can be used as printf's to monitor SMM execution flows. These provide access to SMM code using the UDK's SMM Communication2 Protocol.

For more white box style debugging and testing, the fact that the console resources are managed by the UEFI Boot Service drivers, and subsequently the operating system after ExitBootServices() invocation, means that having a debug print facility is difficult to support.

Also, the use of ASSERTs, which typically result in a jump-to-self condition, are typically used to detect some error conditions on a DEBUG build. On a release build, though, this facility is not available. Even for critical error conditions that are mapped from ASSERTs to dynamic checks, what to do in the case of an error path is a challenging problem for developers. Failing safely is imperative, just as much as making forward progress since a jump-to-self (a "`while (1){}`" condition in the code) can be ascertained via the ITP, but on a release build of the firmware on a production system in the field, this would look like an overall system hang.

We propose a new path to BIOS validation and bug hunting: symbolic execution.

3 Symbolic execution

Symbolic execution [10] [11] [12] is an approach to software testing that searches for interesting, high-quality test cases that will push the software into previously untested regions of the code, or will induce the software to make mistakes like reading past the end of a buffer or dereferencing an invalid pointer. Symbolic execution thus enhances both code coverage and bug hunting.

Symbolic execution works by selecting a set of program variables as "interesting" (perhaps just the input to the function under test), assigning these variables symbolic values, and "executing" the code to construct sets of constraints on these symbolic values. When it encounters a branch that depends on a symbolic value, it forks two new sets of constraints, one in which the branch condition is true and one in which the condition is false. In this way, it builds a symbolic computation tree for the code.

When symbolic execution reaches the end of a computation path (a leaf of the computation tree), it hands the resulting set of constraints to a solver, and produces an actual, concrete test case that will follow this computation path. Continuing in this way, we can (in principal) accumulate sets of input values (test cases) that will together cause the program to execute every line of code (code coverage) and every branch in both directions (branch coverage). The same idea can be used to find inputs that would cause the program to index past the end of an array or dereference an invalid pointer. Or cause an SMM interrupt handler to execute memory outside of SMRAM.

Concolic testing [13] [14] is an approach to symbolic execution that repeatedly executes a program on concrete inputs, but piggybacks symbolic execution on top of the concrete execution via instrumentation (hence the name *concolic* from *concrete* and *symbolic*) . By keeping track of the constraints imposed on symbolic values in the

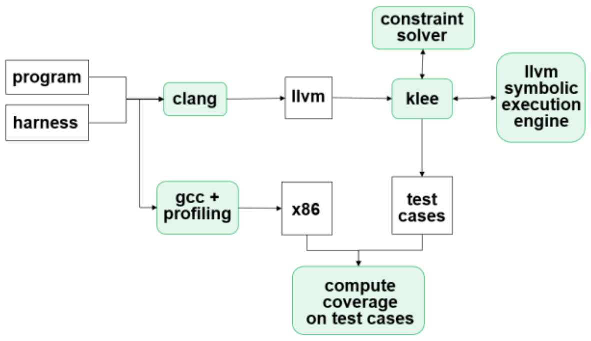

Figure 1: The KLEE work flow

course of a concrete execution, a solver can use these constraints to find new concrete values that will push the computation down a new path.

4 KLEE

KLEE [12] is an open-source implementation of symbolic execution that started at Stanford University and moved to Imperial College. KLEE can be used with any code written in C to generate a minimal set of inputs inducing maximal code coverage. Consider a simple function with type

```
int f(int x);
```

The key to using KLEE is writing a test harness that KLEE can use to explore paths through the code `f(x)`. The primary purpose of the test harness is to identify the variables to make symbolic. The test cases KLEE generates are assignments of values to these symbolic variables. These symbolic variables are the “knobs” available to KLEE as it explores the computation tree, collecting constraints on these symbolic variables that determine a particular computation path through the tree. For a simple function like `f(x)`, the test harness can be as simple as

```
#include <klee/klee.h>
int main(int argc, char *argv[]) {
  int x;
  klee_make_symbolic(x,sizeof(x),”x”);
  f(x);
  return 0;
}
```

The KLEE workflow is illustrated in Figure 1. KLEE works by compiling the source code for the software under test together with the test harness to LLVM [15] with the compiler Clang [16], and running the LLVM on a symbolic execution engine for LLVM written by the KLEE project. LLVM is an aggressive compiler optimization project that performs its optimization on an assembly-level intermediate language called LLVM bitcode. Clang is a compiler that compiles C to LLVM (and then to machine code), and front ends like Clang exist for many languages. We compile the code and the test harness to LLVM and run KLEE to generate test cases. Then we compile the code and the test harness to native code with coverage profiling (e.g., gcov with gcc), and run the profiled code on the test cases to generate coverage information.

KLEE is fast! The idea of using KLEE to look for security vulnerabilities like buffer overflow is almost immediately obvious, and, in fact, a tutorial at BlackHat 2014 [17] made exactly this point and demonstrated to the world how to do it. We have at Intel spent some time using KLEE (and before that an internal implementation of concolic testing) to do exactly this for BIOS and Intel security products, and we have found some limitations to symbolic execution at the source-code level that are hard to overcome.

One problem is the need for C code. Every project like BIOS interacting directly with the hardware includes embedded assembly code. We must either stub out this code or restrict our work to parts of the program that don’t execute code.

The real problem, though, is writing the test harness. The need to write a test harness is just one more obstacle to keep a busy developer from using the tool, no matter how helpful the tool. Automating test harness generation is essential for tool adoption in a production environment. For simple, purely functional code like `f(x)` above, the test harness is easy to generate, but generation gets harder rapidly. Consider the function

```
int f(int x, struct foo *y);
```

The type of y is not so simple, but we can search for the type foo and construct the test harness. Now consider the function

```
int f(int x, void *y);
```

The code gives no indication y’s type. We can search for invocations of the function

```
struct foo y;
f(1, &y);
```

and conclude the actual type to use for the second argument is foo until we discover a second invocation

```
struct bar z;
f(2, &z);
```

and conclude that the second argument is actually a union type. Now consider

```
int f(void);
```

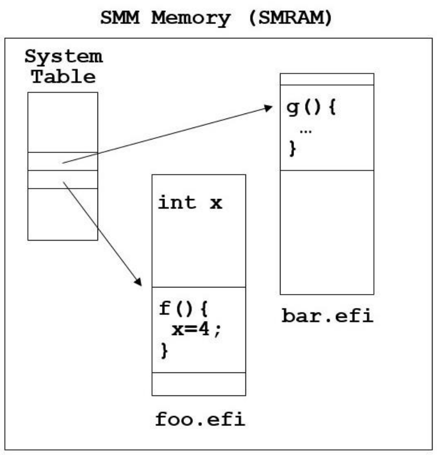

Figure 2: SMRAM is` a protected region of memory containing all SMM code and data.

The function passes all information through the global state. There is no local information to help write the test harness. We can search the body of the function and find

```
extern struct foo y;
y->bar = 1;
```

and build up the portion of the global state that the test harness must model to test the function, but the ease of automating test harness generation is rapidly going downhill.

The fundamental problem is that the test harness must model for the software under test all aspects of the environment that the software depends on. As we have just described, this may not be too hard for libraries or functions near the leaves of the computation tree. For code closer to the root of the computation tree, however, the amount of state to model can be immense. And just knowing the state to model is not enough. We must also know and model all of the assumptions (written and unwritten) about the state variables. We are guaranteed to be flooded with false positives unless we model important assumptions like `x<10` when `is_odd(y)`. Building the test harness, either manually or mechanically, now can be as hard as writing test cases themselves.

For SMM, and more generally for the runtime services BIOS provides to the operating system, the problem of writing a test harness to model the environment of an SMM function is overwhelming to the point of impossible.

All SMM code and data resides in a protected region of memory called SMRAM illustrated in Figure 2. In SMRAM, the primary SMM data structure is a system table that points to, among other things, entry points for

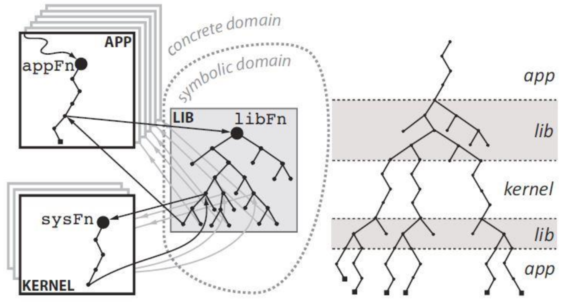

Figure 3: S2E as illustrated in the S2E paper [18].

SMM functions. The code for an SMM function may contain an expression

```
ST->foo->f()
```

invoking a function identified by following a sequence of pointers from the system table. How can we look at code containing a function reference and know what function is actually being invoked? We can't without knowing the system table and the contents of SMRAM. The contents of SMRAM is determined by a long sequence of initializations in which drivers load modules into SMRAM and install into the system table pointers to entry points in the module that implement the functionality being installed by the driver, essentially a form of dynamic linking. The problem of writing a test harness to model the environment of an SMM function essentially becomes modeling the entire initialization process itself, which is almost impossible to automate.

What is frustrating is the fact that everything we need for the model of the SMM environment (the test harness) is sitting in SMRAM after initialization. All the code, all the data, all the layouts. Why can't we just dump the contents of SMRAM, and write a test harness that simply points to an entry point in SMRAM and executes symbolically from there on top of KLEE's symbolic execution engine? S^{2}E is a path to KLEE's symbolic execution engine that lets us do exactly that.

5 S^{2}E

S^{2}E [18] is a tool from École Polytechnique Fédérale de Lausanne (EPFL) built on top of the QEMU virtual machine [19] and the KLEE symbolic execution engine [12]. S^{2}E operates directly on x86 binaries (no source code required) and considers the entire system (application, library, operating system, kernel, device firmware, and so on) to be a single binary program. S^{2}E (which stands for *selective symbolic execution*) lets us specify which parts of this program should be executed concretely and which parts should be executed symbolically, and to switch back and forth between concrete and symbolic execution of the code under test.

Consider the example illustrated in Figure 3 taken from the S^2E paper [18]. Here an application calls a library that calls into the kernel, which returns to the library which returns to the application. Suppose it is only the library that we want to explore. S^2E makes it possible to execute only the library symbolically, shadowing concrete values symbolically in the library, and carefully replacing symbolic values with consistent concrete values in the kernel and application.

S^2E allows us to perform our analysis within a real software stack, with actual user programs running with actual libraries, kernel, drivers, and so on, without building abstract models of the code under test required by a tool like KLEE. And because it operates directly on binaries, we can even study the code under test in the context of proprietary software for which source code is not even available.

S^2E is implemented as an extension of the QEMU virtual machine [19]. QEMU is based on dynamic binary translation. It works by repeatedly fetching basic blocks of the guest code running on the guest architecture and translating them into host code running on the host architecture. It performs this translation with a front end translating the guest instructions into QEMU micro-operations, and a back end translating micro-operations into host instructions. QEMU comes with front and back ends for x86. S^2E simply adds a back end translating micro-operations into LLVM bitcode. When S^2E encounters a basic block to execute concretely, it produces x86 code to run natively. When S^2E encounters a basic block to execute symbolically, it produces LLVM bitcode to execute symbolically on the LLVM symbolic execution engine from KLEE.

The power of S^2E, and the unlimited extensibility of S^2E, rests on its system of plugins. A *plugin* in S^2E can subscribe to a set of *signals* and register *callbacks* for S^2E to run when a signal is raised. For example, one signal is raised when the dynamical binary translator is about to translate an instruction, letting us mark particular instructions as interesting. Another signal is raised when an interesting instruction is about to be executed, letting us intercept (for example) a particular function invocation or return and modify arguments to the function or return values. And another signal is raised when S^2E encounters a branch `x<10` depending a symbolic value `x` and has to branch the current state into two states, one with `x<10` and one with `x≥10`, allowing us to control how S^2E searches the computation tree, perhaps by pruning the `x<10` branch from the computation tree.

An important example of how S^2E facilitates exploration of BIOS code is that we can use a plugin to over approximate the behavior of a device. We can use a plugin to intercept calls to functions accessing the device, throw away the function inputs, and return a symbolic (arbitrary) value from the device. In this way, we can stub out all hardware devices and reduce code exploration to a pure-software problem. If this over approximation results in too many false-positives, our plugin can model the device more accurately.

6 Our approach

Our goal is use S^2E to explore execution paths through SMM interrupt handlers and discover paths in which an interrupt handler tries to access memory outside of the protected region SMRAM for SMM code. We want to generate input test cases that will exercise every branch in the code for test purposes, and induce insecure memory references whenever possible.

To do this, we write a test harness that S^2E can use to explore paths through an interrupt handler. We can either write the test harness as a user-level application that runs from a bash shell on top of an operating system, or as a UEFI application that runs from a UEFI shell on top of a UEFI implementation of BIOS. The test harness has two jobs. First, it must establish the SMRAM region of memory. Second, it must identify a small number of key variables to give symbolic values. Again, these symbolic variables are essentially the "knobs" that S^2E has to play with as it explores the computation tree. It is by varying the values of these symbolic variables that S^2E is able to control the outcomes of branches in the code.

For the test harness to establish the SMRAM region of memory, our approach is illustrated in Figure 4.

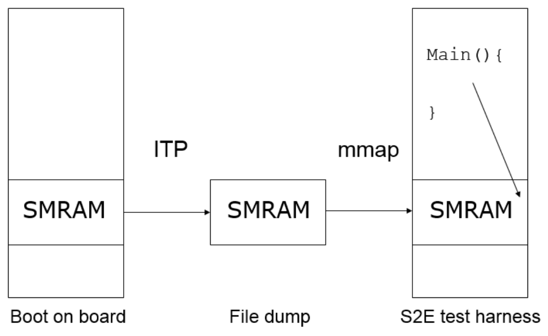

Figure 4: Generating a test harness for an SMM interrupt handler.

First we boot BIOS on a development board to some point where SMRAM has been established (e.g., just before access to SMRAM is locked). Then we dump the

```
// SMRAM image
#define SMRAM_IMAGE "SmramDump.bin"
#define SMRAM_START (0x7b001000)
#define SMRAM_SIZE (0x7b3f6000 - 0x7b001000)
#define HANDLER_ENTRY (0x7B3E6998)

// Handler signature
typedef
  EFI_STATUS EFIAPI SmmVariableHandler(EFI_HANDLE, VOID *, VOID *, UINTN *);

// Handler CommBuffer
#define COMMBUFFERSIZE 100
char CommBuffer[COMMBUFFERSIZE];
UINTN CommBufferSize=COMMBUFFERSIZE;

int main (int argc, char *argv[]) {
  // map the SMRAM image into memory
  FILE *fp = fopen(SMRAM_IMAGE, "r");
  mmap((void *)SMRAM_START,
       SMRAM_SIZE,
       PROT_READ | PROT_EXEC | PROT_WRITE,
       MAP_PRIVATE,
       fileno(fp),
       0);

  // create the symbolic communication buffer
  s2e_make_symbolic(CommBuffer,sizeof(CommBuffer),”CommBuffer”);

  // call the interrupt handler
  ((SmmVariableHandler *)HANDLER_ENTRY)(0,0,CommBuffer,&CommBufferSize);

  // terminate symbolic execution before returning to the shell
  s2e_kill_state(0, “handler done”);
  return 0;
}
```

Figure 5: Test Harness for SMM variable interrupt handler.

contents of SMRAM to a disk file via the In-Target Probe (ITP) [9], a debug tool commercially available from Intel, along with symbol table information including the base and extent of SMRAM and the entry point for the handler we want to exercise. (Alternatively, we could use SerialICE [20] in place of ITP, or we could write a UEFI driver to dump SMRAM in place of both.) The test harness uses this image to establish SMRAM in memory. If the test harness is running as a user-level application, we simply memory map the SMRAM image into the correct location of the user address space with mmap. If the test harness is a UEFI application, we allocate the needed pages in memory and write this image into memory.

For example, consider the SMM variable interrupt handler with interface

```
EFI_STATUS
EFIAPI
SmmVariableHandler (
  IN     EFI_HANDLE DispatchHandle,
  IN     CONST VOID *RegisterContext,
  IN OUT VOID       *CommBuffer,
  IN OUT UINTN      *CommBufferSize
  )
```

The handler communicates with its caller via a communication buffer containing a command to read or write a particular environment variable (or some other variable-related command). This communication buffer is an obvious candidate for symbolic data, since it is the contents of this buffer that drives the behavior of the interrupt handler and is untrusted input to SMM.

The test harness for this handler (ignoring header files, signal handling, etc.) is given in Figure 5. With this test harness we are able to explore the “get variable” functionality of the handler in a few seconds and generate 50 test cases.

The simplicity of the test harness is the primary advantage of using S^2E as the path to the KLEE symbolic execution engine instead of KLEE itself. The test harness for KLEE would need to model the entire environment of SMM. The test harness for S^2E uses SMRAM as the SMM environment, which it is. Writing the test harness now reduces merely to identifying what aspects of the state to make symbolic and jumping to the entry point of the handler.

7 Current results

We have demonstrated that we can boot BIOS on a development board to a point where the SMRAM region of memory containing all the SMM code and data has been established, dump the contents of SMRAM to a disk image, load this image into S^2E , jump to the entry point of an SMM interrupt handler, and begin symbolic execution of the handler to generate high-quality inputs for validation and code coverage We can then replay those test cases on our development board and analyze the code coverage we have achieved for the execution paths we are testing. In the case of the SMM variable interrupt handler, for example, we can generate 4000 test cases in 4 hours and replay them on the development board in 30 minutes.

We have formulated the research problem in terms of open source infrastructure to foster collaboration with industry and academia. We are using an open source development board called MinnowMax that is a compact, affordable, and powerful development board with an open hardware design that allows for endless customization and integration potential [20]. On this board we are running an open source implementation of UEFI. For researchers without access to this infrastructure, we can likely provide the SMRAM image we are using in our own work.

8 Future plans

Our goal is to develop a tool that we can place in the open source to support the enormous UEFI ecosystem that includes both open source and proprietary code.

Our goal is a turn-key, command-line tool that is sufficiently automated that it can be used on a daily basis along with other development tools, like Klocwork [21], but finding concrete instances of security issues that Klocwork may or may not be able to warn about. Given a copy of SMRAM dumped from a development board, a simulator, a virtual platform, or any other emulation environment, and given the base and extent of SMRAM in memory, the tool will scan SMRAM for interrupt handler entry points and generate test harnesses that load SMRAM and jumps to each entry point, and execute each handler symbolically to generate high-quality test cases, including test cases that induce security issues like reading outside of a protected region.

We believe that this approach to symbolic execution — dumping a memory image and executing code in that image symbolically — will apply to many other instances of embedded firmware outside of BIOS in the style of Avatar [23]. In the Internet of Things, simple devices with firmware written and release quickly would be a great target for this approach to bug hunting.

But we face a number of obstacles to this goal.

8.1 Performance

Performance is always an issue. We are looking for a tool that can be run as frequently as a compiler, but we may have to settle for a tool that runs at routine checkpoints. We have, however, already made some progress on this front.

Out-of-the-box performance of S^2E can be improved by a factor of thirty with the correct command line arguments. On small examples like quicksort, S^2E can be made to run just as fast as KLEE, and attains complete code coverage in seconds.

We were also able to address operating system overhead. The most common way to use S^2E is to run the test harness as a user-level application on top of an operating system running on QEMU. We are able to run a test harness as a UEFI shell application on QEMU instead of a user-level application (eliminating the overhead of an entire operating system). On our small examples, we do not see much performance improvement from this optimization, but we expect this to pay dividends on larger, more memory-intensive examples.

8.2 State explosion

State explosion is a known problem for symbolic execution. Given a loop controlled by a symbolic variable that can assume values 0 through 10, symbolic execution will likely generate test cases for all values 0 through 10, even though only the values 0, 1, 9, and 10 are interesting. The literature is filled with heuristics for dealing with this problem, and we will be implementing them.

The S^2E plugin mechanism, however, gives us a particularly modular way to experiment with such heuristics. Because the plugin infrastructure allows us to register interest in state branching — when S^2E splits a symbolic state into two symbolic states, one for each outcome of a branch condition — we have complete control over the choice of states to explore, terminating further symbolic execution from redundant states as heuristics discover them.

8.3 Measuring code coverage

The standard way of measuring coverage with S^{2}E is coverage at the level of machine language, but developers work with source code, so we need to give coverage at the level of source code. The standard way of doing this is to generate the test cases, and then to run the code on the test cases, after compiling the source code with code coverage profiling such as gcov [22] that comes with the gcc compiler. Such profiling usually depends on libc and a file system we don't have with BIOS. Our goal is to generate the test cases with S^{2}E on top of QEMU, but to run the test cases on the development board itself (that is, in the actual product development environment). For BIOS, we need a different approach to code coverage.

We are currently using an Intel product for measuring BIOS code coverage [25]. In the spirit of developing an open source tool, however, we expect also to try experiment with a modification of gcov for embedded systems [23] that writes profiling data to an in-memory data structure instead of to a file system.

8.4 Inducing security vulnerabilities

Our goal is to generate concrete test cases that induce security vulnerabilities whenever they are possible. We are starting with insecure memory references, instances of a handler reading data outside of a safe region and potentially under the control of an attacker.

The S^{2}E plugin mechanism, once again, gives us a modular way to perform such checks. Because the plugin infrastructure allows us to register interest in particular machine instructions, we can examine individual load and store instructions. S^{2}E already comes with a memory checking plugin that makes it possible to trap on every memory reference and check that the address is in range or otherwise satisfies some constraints. We are working to modify this plugin to trap on every memory reference and — instead of checking the address — invoke the constraint solver to ask if there is any assignment of values to the symbolic variables that would cause this address to be out of range (e.g., outside of the protected SMRAM memory range).

8.5 Automation

Our goal is to automate the application of this tool as completely as possible. Remember that one function of a test harness is to identify the variable to treat symbolically. Automating the choice of the symbolic variables, however, may require some annotation or user-supplied information. We may require the user to indicate, for example, that the SMM variable handler communicates with the caller via the communication buffer (and hence that this is a good candidate for symbolic data).

UEFI's modular extensibility, however, may make this annotation less painful than it might otherwise be. UEFI drivers install implementations of interfaces that UEFI refers to as protocols. The complexity of SMM stems from the fact that different systems may use different implementations of the same protocols. But once a protocol is defined, we believe it will be possible to annotate the protocol (the interface) and reuse the same annotation with all implementations of the protocol. In this way, annotation proceeds in a modular fashion, with protocol designers indicating at protocol interface definition what data, when made symbolic, is most likely to explore a lot of the computation tree.

9 Related work

We are not the first to use symbolic execution in the hunt for security bugs.

Work by Davidson *et al* [24] built a tool on top of KLEE to find security bugs in firmware applications written for the MSP430 microcontroller. They find two types of bugs: memory safety violations, such as buffer overruns and out-of-bounds accesses to memory objects like arrays, as well as peripheral-misuse errors in which a firmware writes to a read-only memory location or to locked flash. Work by Corin and Manzano [25] did taint analysis to analyze the security of code by the KLEE symbolic execution engine with a tainting mechanism that tracks information flows of data. Work by Godefroid [26] independent of KLEE addressed our issue of the test harness, and showed that a test harness could be avoided altogether by dynamically mapping the memory footprint of an executable with a virtual machine that traps on memory references, and a configurable memory policy that determines which of the memory references should be treated as inputs. All of this impressive work differs from ours in that our work is addressing run-time services and not stand-alone applications, and the dynamic-linking in the SMRAM layout described in Section 4 that pushed us from KLEE to S^{2}E.

S^{2}E has been used in a number of contexts. Kuznetsov *et al* [27] used an early version of S^{2}E to find bugs in device drivers. They applied their tool (DDT) to several closed-source Microsoft-certified Windows device drivers and discovered 14 serious new bugs. Chipounov *et al* [28] used an early version of S^{2}E to reverse engineer x86 binaries. Their tool (RevNIC) takes a closed-source binary driver, automatically reverse engineers the driver's logic, and synthesizes new device driver code that implements the exact same hardware protocol as the original driver. Zaddach *et al* [29] built a tool Avatar on top of S^{2}E to check security of embedded firmware, and used it to do reverse engineering, vulnerability discovery and hardcoded backdoor detection. In hindsight we might have saved some time by starting with Avatar, since they also run with state dumped from a development board, although security mechanisms surrounding SMM make this quite nontrivial in our case. What impressed us most about this work was their ability to do symbolic execution on top of actual devices, a technique that may be helpful to us in the future.

In other security work based on symbolic execution, Avgerinos *et al* [30] used symbolic execution to analyze 14 open-source projects and successfully generated 16 control flow hijacking exploits. Saxena *et al* [31] built a tool called Kudzu for JavaScript and used it to find finding client-side code injection vulnerabilities. In experiments on 18 live web applications, they automatically discovers two previously unknown vulnerabilities and nine more that were previously found only with a manually-constructed test suite.

10 Conclusion

Ours is a work in progress. We have demonstrated but minimal functionality. But we have demonstrated what we consider to be a new approach to BIOS security validation, using symbolic execution to search for dynamic instances of security vulnerabilities not found by static tools, and doing that symbolic execution directly on top of production binaries taken directly from the development environment. Our approach is to support the open source, academic, and industrial communities in high-quality, secure BIOS implementations. We welcome collaboration in this endeavor.

Acknowledgments

We would not be where we are today without the right help from the right people at the right time. We thank Vitaly Chipounov for help on S^{2}E and critical clarifications on an early draft of this paper, Mark F. Brown on compiler issues, Jordan L. Justen on QEMU and OVMF (UEFI for QEMU); Jaben Carsey on the UEFI shell; Daryl McDaniel on stdlib for UEFI; Jiang A. Liu and Alex Gu on UEFI code coverage enabling; Mike Kinney and Robert P Hale for help and guidance on UEFI; and Michael P. Krau for help obtaining the MinnowMax development board and a getting ITP working. Thanks.

11 References

[1] T. Fox-Brewster, "'Voodoo' Hackers: Stealing Secrets From Snowden's Favorite OS Is Easier Than You'd Think," 18 March 2015. [Online]. Available: http://www.forbes.com/sites/thomasbrewster/2015/03/18/hacking-tails-with-rootkits/.

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

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

[4] R. Wojtczuk and C. Kallenberg, "Attacks on UEFI Security," in *CanSecWest*, Vancouver, British Columbia, March 2015.

[5] V. Zimmer, "UEFI, Open Platforms and the Defender's Dillema," in *CanSecWest*, Vancouver, British Columbia, March 2015.

[6] "Unified Extensible Firmware Interface Forum," [Online]. Available: http://www.uefi.org/.

[7] "Unified Extensible Firmware Interface Specification," Version 2.5, April 2015. [Online]. Available: http://www.uefi.org/sites/default/files/resources/UEFI 2_5.pdf.

[8] "UEFI Platform Initialization Specification (Volumes 1 through 5)," Version 1.4, April 2015. [Online]. Available: http://www.uefi.org/sites/default/files/resources/PI_1_4.zip.

[9] "Intel® In Target Probe (ITP) ITP-XDP 3BR Kit," [Online]. Available: https://designintools.intel.com.

[10] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385--394, 1975.

[11] C. Cadar, V. Ganesh, P. M. Pawloski, D. L. Dill and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th International Conference on Computer and Communications Security (CCS 2006)*, Alexandria, VA, 2006.

[12] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)* , San Diego, CA, December 8-10, 2008.

[13] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, 2005.

[14] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European software*

engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, 2005.

[15] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, March 20014.

[16] "clang: a C language family frontend for LLVM," [Online]. Available: http://clang.llvm.org.

[17] J. Cohen, "Contemporary Automatic Program Analysis," in *BlackHat*, August 2014.

[18] V. Chipounov, V. Kuznetsov and G. Candea, "The S2E Platform: Design, Implementation, and Applications," *ACM Transactions on Computer Systems (TOCS) Special issue ASPLOS 2011*, vol. 30, no. 1, February 2012.

[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005.

[20] "MinnowBoard," [Online]. Available: http://www.minnowboard.org/.

[21] "Klocwork," [Online]. Available: http://www.klocwork.com/.

[22] "gcov—a Test Coverage Program," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. [Accessed 22 May 2015].

[23] H. Blasum, F. Görgen and J. Urban, "Gcov on an embedded system," in *GCC for Research in Embedded and Parallel Systems*, Brasov, Romania, September 2007.

[24] D. Davidson, B. Moench, S. Jha and T. Ristenpart, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *22nd Usenix Security Symposium*, Washington, DC, August 2013.

[25] R. Corin and F. A. Manzano, "Taint analysis of security code in the KLEE symbolic execution engine," in *Proceedings of the 14th international conference on Information and Communications Security (ICICS)*, Hong Kong, China, 2012.

[26] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014.

[27] V. Kuznetsov, V. Chipounov and G. Candea, "Testing Closed-Source Binary Device Drivers with DDT," in *USENIX Annual Technical Conference (USENIX)*, Boston, MA, June 2010.

[28] V. Chipounov and G. Candea, "Enabling Sophisticated Analysis of x86 Binaries with RevGen," in *7th Workshop on Hot Topics in System Dependability (HotDep)*, Hong Kong, China, June 2011.

[29] J. Zaddach, L. Bruno, A. Francillon and D. Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *21th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014 .

[30] T. Avgerinos, S. K. Cha, B. L. T. Hao and D. Brumley, "AEG: Automatic Exploit Generation," in *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[31] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and S. Dawn, "A Symbolic Execution Framework For JavaScript," in *31st IEEE Symposium on Security and Privacy*, Oakland, CA, 2010.

[32] "Code coverage component of the Intel® Intelligent Test System (Intel® ITS)," [Online]. Available: https://designintools.intel.com.

[33] "SerialICE," [Online]. Available: http://www.serialice.com.