Improving Mac OS X Security Through Gray Box Fuzzing Technique

Stefano Bianchi Mazzone
ste@security.di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

Mattia Pagnozzi
mattia@security.di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

Aristide Fattori
aristide@security.di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

Alessandro Reina
reina@security.di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

Andrea Lanzi
andrew@security.di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

Danilo Bruschi
bruschi@di.unimi.it
Dipartimento di Informatica
Università degli Studi di Milano

ABSTRACT

The kernel is the core of any operating system, and its security is of vital importance. A vulnerability, in any of its parts, compromises the whole system security model. Unprivileged users that find such vulnerabilities can easily crash the attacked system, or obtain administration privileges. In this paper we propose LynxFuzzer, a framework to test kernel extensions, i.e., the dynamically loadable components of Mac OS X kernel. To overcome the challenges posed by interacting with kernel-level software, LynxFuzzer includes a bare-metal hardware-assisted hypervisor, that allows to *seamlessly* inspect the state of a running kernel and its components. We implemented and evaluated LynxFuzzer on Mac OS X Mountain Lion and we obtained unexpected results: we indivuated 6 bugs in 17 kernel extensions we tested, thus proving the usefulness and effectiveness of our framework.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Reliability – Verification – Security and Protection

1. INTRODUCTION

Kernel security is of vital importance for a system. A vulnerability, in any of its parts, may compromise the security model of the whole system. Unprivileged users that find such vulnerabilities can easily crash the attacked system, or obtain administration privileges. Unfortunately, kernels are an evermore attractive target for attackers, and the number of kernel vulnerabilities is rising at an alarming rate [18]. Looking for vulnerabilities in kernel-level code is a daunting task, because of its many intricacies. Indeed, modern kernels are extremely complex and have many subsystems, possibly developed by third parties. Often, such components are not as secure as the kernel, because of the lack of testing, and their typical closed-source nature [1, 2, 19, 3]. Furthermore, kernels have countless entry points for user data. System calls, file systems, and network connections, among others, allow user-fed data to reach important code paths in the kernel. If a bug is found in such paths, it can lead to vulnerabilities that compromise the entire system security. Testing the user-to-kernel interface is really important because it can reveal a potential privilege escalation vulnerability that represents one of the main targets attack nowadays [20]. Windows and Linux user-to-kernel interfaces have been deeply analyzed and many tools exist for their testing and verification. On Mac OS X, the user-to-kernel communication approach used by kernel extensions [5] is relatively young and has not been extensively analyzed before. Furthermore, the number of OS X adoptions is growing at a continuously rising pace and this will soon attract the attention from cyber-criminals.

In this paper, we present the design and implementation of LynxFuzzer, a framework to automatically find vulnerabilities in *kernel extensions* (kexts), i.e., dynamically loadable components of the Mac OS X kernel. Kexts are built adhering to the *IOKit framework* [5], the official toolkit for creating OS X kernel extensions (also used in the iOS environment). LynxFuzzer uses a gray box fuzzing approach and adopts dynamic test-case generation. This means that it *automatically* extracts information from the target extensions and uses them to *dynamically* adapt its input generation techniques. More precisely, we implemented three different fuzzing engines inside LynxFuzzer: a simple *generation* engine that produces pseudo-random inputs that only respect kext-defined constraints, a *mutation* engine that builds input data from previously *sniffed* valid inputs, and an *evolution-based* engine that leverages evolutionary algorithms [6] and uses code coverage level as a main validating feature.

We decided to build LynxFuzzer hypervisor component on top of the open source hardware-assisted virtualization framework HyperDbg [13] mainly because of the many difficulties of running Mac OS X inside commonly available virtualization software. In fact Apple is well-known for its custom hardware (e.g., the Magic Mouse), which is not often emulated by common hypervisors. This shortcoming would prevent LynxFuzzer from testing some of the drivers in OS

EuroSec '14 April 14–17, 2014, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2592791.2592793 ...$15.00.

X. Moreover an hardware-assisted environment assures the possibility of gathering dynamic information about the state of the machine during the crash of the system. In summary our work contributes the following:

- We design LynxFuzzer, a framework that is able to automatically find vulnerabilities in *kernel extensions* (kexts), i.e., dynamically loadable components of the Mac OS X kernel. We have tackled serveral implementation challenges to make an efficient fuzzing system.
- We devise a set of efficient and trasparent techniques to automatically extract the fuzzing input constraints that allow the system to reduce the inputs searching space and to efficiently fuzz the Mac OS X user-to-kernel communication interface.
- We developed a prototype of the fuzzying system, and performed an experimental evaluation on serveral kext extensions. Our experiments show that the system is able to individuate 6 bugs in 17 kexts we analyzed. Two of these bugs have already been patched in OS X 10.9 (Maverick) and have been assigned the following CVE-ID by Apple: CVE-2013-5166 and CVE-2013-5192.

2. IOKIT FUNDAMENTALS

In this section we will give an overview of the IOKit framework, as this is basic for the understanding of LynxFuzzer.

Mac OS X is an operating system developed by Apple and currently adopted on Apple computers. Among its many components, the Kernel contains one that is particularly relevant for our work: the IOKit. IOKit is a *"a collection of system frameworks, libraries, tools, and other resources for creating device drivers in OS X"* [5]. It offers an object oriented environment and a number of abstractions that make the (commonly painful) task of writing kernel components much easier and "user space friendly". The IOKit framework contains many abstractions and components, and we describe those relevant to our work in the following of this section.

```
const IOExternalMethodDispatch
UserClient::sMethods[kNumberOfMethods] = {
{
 /* kMyScalarIStructIMethod */
 (IOExternalMethodAction) &UserClient::sScalarIStructI,
 /* One scalar input value */
 1,
 /* The size of the input struct */
 sizeof(MySampleStruct),
 /* No scalar output values */
 0,
 /* No struct output value */
 0
},
/* ... */
}
```

Figure 1: A sample dispatch table.

A fundamental component of every OS is the communication between user- and kernel-space components. Windows and Linux offer such functionality through system calls and special virtual files (e.g., `/dev/urandom`). IOKit supports both techniques, but also adds a novel and much more complex mechanism, named *DeviceInterface* [17, 5].

To leverage this mechanism, Kernel extensions [21] (i.e. drivers) define a set of methods that can be invoked by user-space programs. Such methods have limitations in terms of number and type of data that can be received by and returned to the caller. The list of these methods and constraints on their parameters are stored into a structure of utmost importance (both for IOKit and for our fuzzing framework): the *dispatch table*. Each kext can define one or more dispatch table. Each table is an array of structures, each one containing a function pointer, allowed input and output values, as well as the number and size of the values that can be accepted (or that are returned) by the method. If the size of an input structure is not known *a priori*, the check can be demanded to the receiving method, by specifying a size of `0xffffffff` in the table. It is possible to have, for any driver, more than one dispatch table. IOKit, indeed, allows kexts to offer multiple interfaces to user-space programs at the same time. Each kext must also define one custom subclass of *UserClient* for each interface. Instances of these subclasses are then loaded in the kernel memory along with the kext (Figure 2). Every *UserClient* object contains the dispatch table corresponding to the interface it offers.

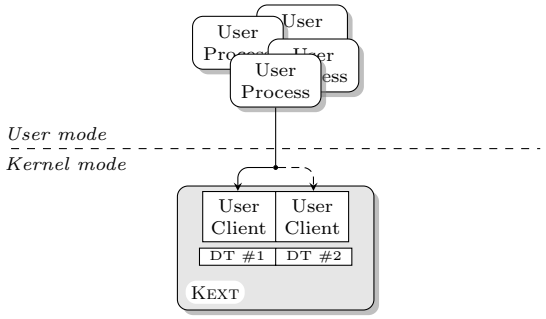

Figure 2: Invoking a kext's method.

User-space programs can invoke kexts methods via `IoConnectCallMethod()`, if the methods are specified in one of the kext dispatch tables. To be able to do so, however, the program must first find the specific instance it wants to talk to. To illustrate how this happens, we must first introduce one more IOKit abstraction, the *IOService*. Each IOKit device driver is an object that inherits from the IOService class and a kext may contain different IOService objects at the same time. Let us consider an example. There are typically many USB devices connected to a computer and each one needs its own driver. Such drivers are all contained in the `IOUSBFamily` kext, and each of them is a specific IOService subclass (*service*, for short) for a device. When a user-space process wants to communicate with one device, as we mentioned above, it establishes a mach connection [17] to IOKit and searches for the specific service associated to the device. This process is called *Device Matching* [5].

Once the communication channel is established and the service is found, the user-space program uses `IoConnectCallMethod()` to invoke the desired method. This transfers the control to the IOKit framework that, before actually executing the target method, performs a set of operations. At first, it retrieves the dispatch table entry from the *UserClient* object. The entry is then passed to the `externalMethod()` function along with other parameters that allow to perform the actual invocation of the kext method. This, however, happens *only* if the parameters correspond to what is specified by the dispatch table entry, otherwise the invocation is blocked.

The whole IOKit input control mechanism offers an additional protection layer, with respect to common mechanisms such as `ioctl`, because checks are performed before parameters are even moved from user- to kernel-space. Obviously, all these constraints make the fuzzing process quite complex. It is almost useless to just fuzz functions of a kext with completely random parameter sizes, as most invocations would be discarded by checks performed by IOKit. As we will see in the next section, one of the key aspects of our fuzzer is its ability *automatically* extract constraints on parameters from the target and *dynamically* adapt its fuzzing techniques to get better efficiency.

3. LYNXFUZZER

The purpose of our fuzzer is to trigger bugs in kext code that can be reached by user-space programs, by means of the IOKit framework. A bug triggerable by user-space, indeed, may allow un-privileged users to crash the system, or even obtain arbitrary kernel-level code execution, which typically leads to privilege escalation attacks [20]. Furthermore, we decided to focus our efforts on the *DeviceInterface* boundary-crossing mechanism, since it became the *de facto* user-to-kernel communication standard for OS X kernel extensions.

As it should be clear from what we described in section 2, there are many constraints that must be considered when invoking a kext method and such constraints are specific to each kext. Knowing constraints contained in the dispatch table allows us to reduce the possible fuzzing space to what is actually accepted by the kext, thus gaining much more efficiency. Thus, we designed LynxFuzzer so that it can extract these information in a completely automatic fashion and fuzz them autonomously. Furthermore, the automation of our fuzzing infrastructure is not limited to this. Indeed, we are able to extract valid input vectors used in non-artificial interactions between user- and kernel-level components. Such inputs are used as a basis to elaborate new inputs aimed at improving the fuzzing strategy.

An overview of LynxFuzzer architecture is depicted in Figure 3. Our framework has two main components: one that resides in user-space and consists of 4 sub-components, and one built on top of our hypervisor analysis framework. Figure 3 also reports the main interactions between LynxFuzzer internal components. The *tracer* interacts with the hypervisor to identify where the dispatch tables of the target are (1, 2). Once discovered, the hypervisor retrieves them from the kernel memory and sends them back to the tracer that stores them into the *data manager* for later use (2, 3). The sniffer uses such information to *intercept* non-artificial `IoConnectCall()` invocations and gather a set of valid inputs (4, 5). Finally, the fuzzer components starts invoking the target methods with custom parameters (6), waiting for an eventual panic (7). Depending on the selected fuzzing engine, the fuzzer may use valid inputs precedently stored in the data manager to generate new inputs or leverage coverage information (8).

3.1 Tracer

The tracer is the first component of LynxFuzzer to be executed, as its task is finding out *what to fuzz*, i.e., it must identify which of the target methods can be invoked. This information is contained into the dispatch tables of the target kext. Being able to locate the dispatch table of a kext, however, is not easy, since IOKit uses a number of abstraction layers to hide such information to user-space programs. We devised a solution by observing that, whenever a user-space program invokes `IoConnectCallMethod()`, the IOKit will invoke its `externalMethod()` function. Thus, we proceed as follows. LynxFuzzer hypervisor sets a breakpoint on the `externalMethod()` function and intercepts whenever it is executed. Once the trap is set, the tracer issues a request to the target kext with a `selector` value of 0. When the hypervisor intercepts the resulting `externalMethod()` execution, it extracts the base of the dispatch table, uses it to dump the whole table, and eventually returns it to the tracer component. Finally, the tracer stores the extracted dispatch table into the data manager, so that other components can leverage this information for their operations.

Note that the size of the dispatch table is not known *a priori*, nor is contained in the parameters of the trapped function. To solve this problem, LynxFuzzer leverages the structured nature of such tables to infer how many entries it has and dump it. Indeed, each table entry is formed by: a function pointer, which must reside in the memory area of the target, and 4 consecutive integers, two of which must fall in the [0:16) range.

3.2 Sniffer

Beside the checks performed by IOKit, the kext itself could implement constraints on input values. Thus, LynxFuzzer includes a *sniffer* component, that is able to intercept real executions (i.e., not artificially triggered by our framework) of the target methods and to extract their parameters. To do so, we once again leverage LynxFuzzer hypervisor component that is able to *transparently* and *seamlessly* intercept whenever an interesting function is executed, and dump its parameters, by inspecting the target kext memory.

In particular, the hypervisor places a transparent breakpoint on the `externalMethod()` function, whose parameters contain enough information to retrieve valid inputs. Indeed, the hypervisor uses the `dispatch` argument to discriminate which kext is the target of the intercepted invocation and `selector` to understand the method being called. Valid inputs are extracted from `IOExternalMethodArguments` structure containing the actual parameters that will be passed to the callee. Such data structure contains fields that allow to precisely infer the number and size of input parameters that will be stored into the data manager.

3.3 Fuzzer

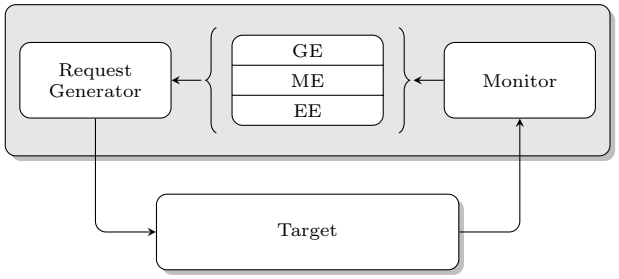

Figure 4: Fuzzer sub-components.

The fuzzer is the main component of LynxFuzzer. After the tracer and the sniffer obtained all the ancillary information needed to properly fuzz one (or more) kext, the fuzzer

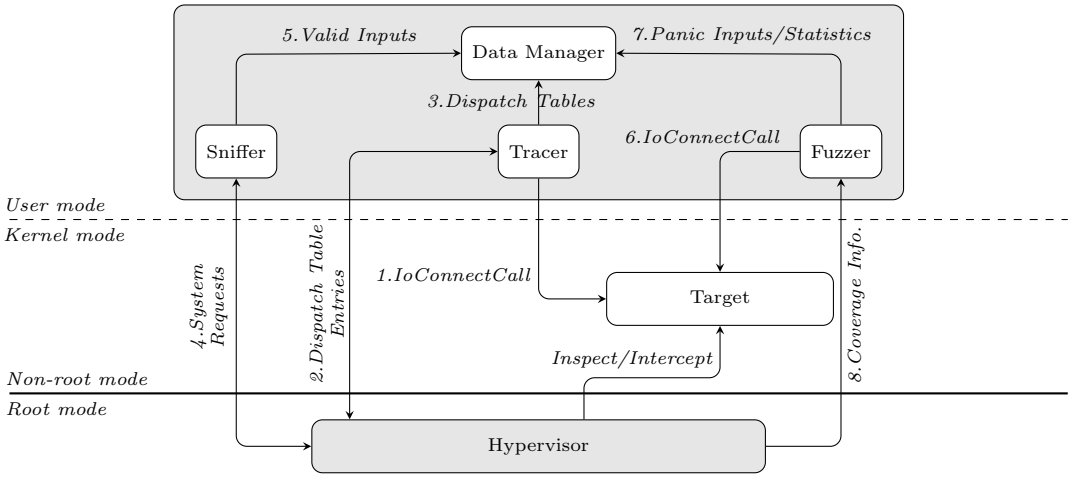

Figure 3: Architecture of LynxFuzzer and interactions between its components (gray areas).

creates test cases (i.e., set of inputs) for the kext methods and invoke them through the IOKit device interface. Figure 4 reports the inner architecture of the component, that has *three* main parts: a request generator, a set of fuzzing engines and a monitor.

The *request generator* is an extremely versatile component: it must operate independently from the target kext *and* the selected fuzzing engine. In a typical execution, it receives a test case from the engine, checks that it respects what is specified in the dispatch table entry of the target method and properly crafts the argument for an `IoConnectCallMethod()` that eventually executes the target method in the kext.

If the test case does not cause a crash, the kext sends back an answer that is received by the *monitor*. Depending on the received answer and on the engine currently in use, the monitor may decide to alter the engine so that the next test case will depend on the result of the previous one. As we will see later in this section, both the *mutation engine* and the *evolution engine* leverage this information.

LynxFuzzer, furthermore, implements the concept of *session-based* fuzzing: we do not save just the request that *triggers* the bug, but we record *every* request that we make from the beginning of the fuzzing session. This practice is common when fuzzing stateful network protocols [8], but is also useful in our scenario. There are indeed kexts that maintain a "state" that is changed by a number of different fuzzing requests until an invalid state is reached and a bug is triggered. For this reason, recording *communication sessions*, instead of single requests, greatly improves the reproducibility of a bug. To record communication sessions between the fuzzer and a target kext, every request is stored in the data manager. Finally, the fuzzing engine is responsible for the production of input values (or *input vectors*) that will be used by the request generator. Details of the three different engines that we implemented in LynxFuzzer are described in the following sections.

3.3.1 Generation Engine (GE)

This is the simplest and quickest engine of the three. Its generation process may be summed up as follows. At first, it builds data structures that can contain the input for the target method. Then, it generates pseudo-random inputs and fill the structure that are then sent to the target, invoking the method through `IoConnectCallMethod()`. If the system does not crash, then the procedure begins anew.

3.3.2 Mutation Engine (ME)

This second fuzzing approach follows a principle that is the opposite of the previous one: every new input is generated from valid inputs collected by the sniffer component. The fuzzing process is roughly made of the following steps. Valid inputs that were previously gathered by the sniffer are *mutated* with different functions. Then, request containing such forged inputs are sent to the target method. If the system does not crash, the monitor checks the response of the kext, possibly excluding values that caused the kext to return an error from next mutations. This greatly increases the efficiency of the fuzzer, in the case of inputs structures with a variable size, because it gradually eliminates those that are not accepted because of checks performed *in* the code of the target method. The set of mutation functions used by this engine includes: *bit flipping*, *byte flipping*, *byte swapping* and *size change*.

3.3.3 Evolution Engine (EE)

The evolution engine tries to overcome the limitations of the previous ones. In an effort to reduce the use of pseudo-randomness, it leverages concepts of evolutionary algorithms to generate new inputs.

The hearth of any evolutionary algorithm is the *fitness* function, that depicts the fittest elements that will contribute to build new generations. In LynxFuzzer, we devised two different fitness functions: one that measures the code coverage of an input vector and one that measures the distance of an input from an ideal target vector (input that crashes the system). In the first case, we strive to create a set of input vectors that can give us the best code-coverage rate possible. The second, on the other hand, is useful when we want to individuate inputs similar to a given one (e.g., a vector that is known to trigger a bug in the target).

Code coverage analysis. Our code coverage analysis method works as follows. Before starting to invoke kext methods,

Kext	Code Coverage Perc.		
	Methods	Estimation	Full
IOUSBFamily	64.8%	61.1%	33.7%
IOHIDFamily	86.4%	69.9%	11.4%
SimpleDriver	96.6%	77.3%	34.3%
IOSurface	76.6%	58.8%	18.5%
AppleUSBHub	86.9%	54.5%	37.5%

Table 1: Coverage analysis results.

the fuzzer component communicates to the hypervisor (via hypercall) the range of code pages of the kext. The hypervisor removes the `EXEC` permission from the EPT entries corresponding to received memory ranges. As soon as the kext tries to execute code in such pages, it triggers an EPT violation and the hypervisor keeps track of the instruction that caused it. To proceed, it restores `EXEC` permission on the faulting page and configures the guest to perform a *single-step*. When the hypervisor gets the control back, because of the resulting debug exception, it removes the permission, so that the next instruction will violate again. Once the fuzzed methods returns, the fuzzer sends another hypercall to disable the tracing. The hypervisor stores collected information into a buffer in the fuzzer address space so that it can be used by the user-space component to calculate the code coverage corresponding to the invocation.

4. EXPERIMENTAL EVALUATION

This section presents the result of the experiment we conducted to evaluate the effectiveness of LynxFuzzer. To this purpose, we used the fuzzer to exercise a set of 17 different Kernel Extensions and we found 6 bugs in them[1]. More detailedly, 2 of the 6 have already been patched in OS X 10.9 (Maverick) and have been assigned the following CVE-ID by Apple: CVE-2013-5166 and CVE-2013-5192 [4]. The remaining 4 are still unpatched and will be most likely addressed in the next releases of OS X.

All the experiments were conducted on a Mac OS X 10.8.2 system (Mountain Lion), installed on Apple hardware (Intel i5 CPU and 12GB of RAM). Thanks to the adoption of kernel-security measures in OS X, none of the bugs we identified can be easily exploited to perform a privilege escalation attack.

A metric that is usually associated with efficiency of a fuzzer, is the *code coverage* level. However, as also stated in [14], such metric may not be extremely significant: a fuzzer could even reach 100% code coverage of the analyzed code, but yet fail at finding a bug. Since it is customary to report such information, however, we conducted an experiment to calculate the code coverage level of LynxFuzzer.

Even if our hypervisor component can easily keep track of every instruction that is executed in a given time-frame (e.g., during the fuzzing), giving a *precise* measure of the coverage of a kext is quite challenging. To give an estimation of the amount of code that can be reached from methods exported in the dispatch table we use, once again, a hybrid static-dynamic technique. First, we statically count the instructions of the exported methods and individuate all the control transfer instructions (CTI). Then, for each CTI, if the target is another method of the same kext, we add its instructions to the overall count.

[1]Obviously, all the bugs were reported to Apple or the appropriate vendor.

Kext	No. of Inst.	Rps w/o Tracing	Rps w/ Tracing	Overhead
AppleSMCLMU	1890	668.28	385.90	1.73x
AppleMikeyDriver	17457	517.54	283.28	1.83x
AppleHDAController	12294	591.82	273.77	2.16x
IOHDAFamily	3376	634.17	278.41	2.28x
AppleSMC	5608	651.99	284.31	2.29x
IOSurface	8866	434.84	112.44	3.87x
AppleHDA	101532	635.39	164.20	3.87x
IOUSBFamily	49920	204.89	42.76	4.79x
SimpleDriver	2261	424.91	74.01	5.74x
IOHIDFamily	63915	278.00	46.44	5.99x
Overall	—	—	3.45x	

Table 2: Overhead of the code coverage analysis.

Unfortunately, this is not enough because, due to their object-oriented nature, kexts include a high number of *indirect* CTIs, that cannot be followed statically. For such instructions, we revert to *dynamic* analysis: we modified LynxFuzzer code coverage analysis module so that it dumps the target of every indirect CTI executed in the kext code. If the target corresponds to a method of the kext that had not been deemed reachable by the static analysis, then we update the instructions count with the newly discovered method.

Table 1 reports the code coverage results of a subset of the fuzzed kexts. Under the "Coverage" column we report three different code coverage percentages, respectively calculated over: the number of instructions of exported methods, the static/dynamic estimation we just described and, finally, the overall number of instructions contained in the kext.

We also evaluated the overhead introduced by our code coverage method that we described in Section 3.3. To measure it, we run the generation engine on each method of 10 different kexts, with and without the code coverage analysis enabled, and measured the consequent decrease in terms of requests per second the kext was able to serve. For the sake of precision, measurements were repeated 10 times for each module and results were averaged. The average overhead is 3.45x, with a best and worst case of, respectively, 1.73x and 5.99x. Detailed results of this evaluation are reported in Table 2. As we can see, we pay a high price for obtaining full precision without modifying the target, yet this is a much lower overhead if compared with other techniques [22].

Finally, during our experiments we also measured the efficacy of each input generator engines to discover bugs. In particular we show that all the engines utilized were able to discover the bugs with some difference in terms of performance. In particular the `Evolution Engine` with the code coverage-based fitness function was faster than the other two, while the *Generation Engine* was the slowest one.

5. RELATED WORK

The closest work to LynxFuzzer is found in an online presentation by Xiaobo and Hao [10] that briefly analyzes the possibility of fuzzing OS X kernel extensions through *DeviceInterface*. Unfortunately, a deep comparison between such work and LynxFuzzer is hindered by the lack of details of the former. The depicted approach appears to be largely manual and there is neither an analysis of performances nor results of conducted fuzzing activities to compare.

Another similar, yet extremely more specific, approach is

presented by Keil and Kolbitsch [15] who propose a stateful fuzzer for 802.11 device drivers. Beside its usage of stateful fuzzing, LynxFuzzer is quite different from this work, since it addresses generic kernel extensions and leverages hardware-assisted virtualization instead of emulation.

SLAM is a model checking engine whose goal is to automatically verify if a program correctly uses external libraries [7, 12]. On top of such engine, Microsoft created Static Driver Verifier, a tool to automatically analyze the source code of Windows device drivers and determine whether they correctly interact with the Windows kernel. SLAM has many limitations, if compared to LynxFuzzer. First, it requires the source code of the drivers it analyzes. Second, its scope is limited to the correctness of interactions with known libraries. Finally, users of SLAM must feed it with manually created rules that the tool will check for the verification, thus requiring some manual effort.

S^2E [11] is a binary analysis framework that implements the idea of *selective symbolic execution*, a method to minimize the amount of code that needs to be executed symbolically. Even if remarkably useful, for example we could use it to simbolically execute every hardware-interacting part of a kext, S^2E has some drawbacks. First, the runtime overhead of 6x for concrete execution and 78x for symbolic execution is extremely high. Second, it is strongly binded to QEMU and, as we stated before, running Mac OS X inside an emulator is not trivial.

Dowser is probably the most advanced approach to fuzzing targeted at finding buffer overflows [14]. It uses a mix of static program analysis, concolic execution, and taint tracking to automatically steer the execution of a program to interesting locations, more likely to contain a buffer overflow. In particular, the observation behind Dowser is that there are particular sets of instructions that are more error-prone than others (e.g., pointer arithmetic). Dowser is extremely powerful, and LynxFuzzer could benefit by adopting a similar approach to its fuzzing activities. Unfortunately, our approach is aimed at testing closed source software, while Dowser relies on the source code of the target to locate interesting instructions.

Finally EXE [9] and SmartFuzzer [16] are similar to LynxFuzzer, as they both use a combination of static and dynamic analysis to automatically generate inputs vector that can efficiently exercise user-space applications.

6. CONCLUSION

LynxFuzzer is a fuzzing framework for Mac OS X kernel extensions. LynxFuzzer leverages hardware-assisted virtualization and three different input-generation engines to discover bugs that could lead unprivileged users to crash the machine or to attempt privilege-escalation attacks. We implement a prototype and we show in our experiments the efficiency and efficacy of our system. LynxFuzzer was able to discover bugs in 6 of them in 17 kernel extensions of Mac OS X Mountain Lion.

7. REFERENCES

[1] CVE-2010-1794, 2010. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-1794`.

[2] CVE-2013-0109, 2013. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0109`.

[3] CVE-2013-0981, 2013. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0981`.

[4] Apple. About the security content of OS X Mavericks v10.9, Oct. 2013. `http://support.apple.com/kb/HT6011`.

[5] Apple Inc. I/O Kit Fundamentals. `https://developer.apple.com`.

[6] D. Ashlock. *Evolutionary computation for modeling and optimization*. Springer Science+ Business Media, 2006.

[7] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Integrated formal methods*, pages 1–20. Springer, 2004.

[8] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: toward a stateful network protocol fuzzer. In *Information Security*, pages 343–358. Springer, 2006.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[10] X. H. Chen Xiaobo. Find Your Own iOS Kernel Bug, 2012.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, California, USA, 2011.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[13] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, September 2010.

[14] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of 22nd USENIX Security Symposium*, Washington, DC, USA, 2013.

[15] S. Keil and C. Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[16] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari. A smart fuzzer for x86 executables. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, 2007.

[17] J. Levin. *Mac OS X and IOS Internals: To the Apple's Core*. Wrox, 2012.

[18] NIST. CVE and CCE statistics. `http://web.nvd.nist.gov/view/vuln/statistics`.

[19] NIST. CVE-2013-0976, 2013. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0976`.

[20] E. Perla and M. Oldani. *A guide to kernel exploitation: attacking the core*. Syngress, 2010.

[21] A. Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006.

[22] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.