École Polytechnique Fédérale de Lausanne

Arm Wrestling: porting the Retrowrite project to the ARM architecture

by Luca Di Bartolomeo

Master Thesis

Approved by the Examining Committee:

Prof. Mathias Payer (EPFL)
Thesis Advisor

Prof. Kenny Paterson (ETH)
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

November 4, 2020

No matter where you go, everyone is connected.

— Serial Experiments Lain

Dedicated to my parents, my sister Sara, my dear Giulia, to my friends back in Rome and to my roommates Matteo and Filippo who all inspired me and kept up with my constant complaining.

Thanks!

Acknowledgments

I would like to thank my advisor, Prof. Mathias Payer, for his support, guidance, and for trusting me by assigning me this inspiring project. I admire him a lot and I wish all the best for him, and in particular I hope that I can work on many other projects with him.

I would also like to thank his research group, HexHive, as I always found myself very welcome there, even if I could visit them only once a week. Those have been six very happy months in which I learned quite a lot of things, and I have to thank prof. Prof. Payer and his doctorate students and researches for it. I wish them all to have a very succesfull career, and I hope we can continue to work together in the future!

Special thanks goes to my family and my friends in Rome. Their support was always available, and it has always been a huge pleasure to visit them once in a while in Italy. I also need to thank my S.O. Giulia, I felt she was always behind my back, keeping a good check on my mental sanity during the worst times of the outbreak. My roommates too, Matteo and Filippo, deserve a mention here, as their patience and their rubber duck debugging skills proved to be fundamental during some nasty debugging sessions.

Finally, I would also like to mention my CTF team, Flagbot, that made me spend so many weekends without going out but ultimately lead me to meet so many new interesting people, and the teams I had played with occasionally, namely polyglots and TRX. Thanks!

Lausanne, November 4, 2020 Luca Di Bartolomeo

Abstract

While there were good recent attempts, (TODO: link to PinePhone, System76, and similar projects) using only open-source software is particularly hard, and especially on the mobile phone market even the most determined users are often forced to use closed-source ARM libraries or modules. Those often run at privileges higher than we might want (e.g. manufacturer specific kernel modules, TODO: link to some), and are also hard to audit for vulnerabilities.

Many existing tools were developed to improve the auditability of closed source programs, especially aimed at helping the fuzzing process, with approaches such as implementing AddressSanitizer (a compiler pass only available with the source code) through dynamic instrumentation. However, even state-of-the-art dynamic instrumentation engines incur in prohibitive runtime overhead (10x and more).

In this thesis, we would like to show that symbolization for ARM binaries is a viable alternative to existing approaches, that has less flexibility (only works on C, position independent binaries) but has negligible overhead compared to compiled source code. We present the ARM port of `Retrowrite`, an existing static binary rewriter for x86 executables, which implements the symbolization engine and the memory sanitization instrumentation.

While `Retrowrite` is not the only binary rewriter for x86, there are very few that are aimed at the ARM64 architecture. In particular, most of them resort to lifting to an intermediate IR (achieving more flexibility but losing fine-grained control over the instrumented instructions) or use approaches like trampolines which make the whole problem much easier but also introduce a noticeable overhead. TODO: add a conclusion to this abstract, like "our proof of work implementation of retrowrite is xx% fastern than QEMU and etc..."

Contents

Chapter 1

Introduction

Security researchers routinely find vulnerabilities in major desktop OSes, but recently a lot of focus has shifted on the analysis of mobile operating systems, including Android. More than 40 CVEs for Android were published just in December of 2019 [1]. This big focus on the analysis of Android has certainly a lot to do on with the rise of the mobile phone market in the latest years, but also with the fact that the source code of Android is (mostly) open and available for everyone to dig through and debug.

Still, digging through the code is not the only way of auditing a codebase, in fact a very effective technique as of today is the process of *fuzzing*. It is taking a lot of traction because setting up a fuzzing campaign, even a large scale one, is relatively easy for beginners too. The main advantages of fuzzing compared to other automatic vulnerability discovery techniques is that you do not need to have a deep knowledge of the fuzzer engine to have a successful run, you just need to select the initial input seed corpus, and implement a fuzzing harness. Moreover, the approach is extremely easy to parallelize, so it is not uncommon for companies to setup large server farms with thousands of CPU cores dedicated to fuzzing.

Having the source code available while fuzzing immensely helps the fuzzer engine thanks to coverage information, with which the fuzzer can prune inputs much more accurately compared to *black-box* fuzzing without the source code. Furthermore, the presence of source code gives

[1] `https://source.android.com/security/bulletin/2019-12-01`

the opportunity to apply other instrumentation passes, which can help the fuzzer even more. For instance, memory sanitization helps the fuzzer find memory corruption bugs as soon as they are introduced without needing to wait for a crash.

Unfortunately, in spite of the increasing popularity of open source software, closed source modules and binaries are still much more common: on desktop, 90% of the operating systems are closed source, and even who runs an open-source alternative usually still has many closed source components (e.g., Skype, Nvidia drivers, various DRM components in browsers, etc.). This holds true even in the phone market: while 75% [12] of mobile users use Android or other open-source operating systems, virtually all of those devices have proprietary components already installed at the moment of purchase. Most commonly there are Google proprietary libraries like the Play API, manufacturer custom modules and firmwares (for hardware peripherals such as Bluetooth or WWAN connectivity), and much more. In many instances, those closed-source modules are installed as kernel modules or run with root privileges. To make things worse, they are often are exposed to the network, creating opportunities for malware or other bad actors to take control of mobile devices.

When considering about fuzzing closed source binaries, the main difficulties that arise are due to the fact that the compilation process discards a large amount of information that could be very useful while hunting bugs, such as type information, control flow, array boundaries, scalars/references distinction, and much more. Still, there are techniques that can help instrument a binary and fuzz it in those conditions, and they all fall into a common category called *Binary rewriting*.

Binary rewriting techniques can be divided into two main approaches: *dynamic* rewriting and *static* rewriting. Dynamic rewriting involves running the target binary in a controlled execution alongside the rewriter engine, which instruments the target binary on the fly. The advantage of this approach is that at runtime the rewriter is able to recover some of the information that was lost during compilation (the most important one being the distinction between scalars and references), and so do not need to rely on heuristics, and can support a very wide variety of binaries (even packed or self-modifying binaries). However, the trade-off is the overhead introduced in the process: dynamic rewriters usually have between 10x and 100x of overhead [9], which has a great impact on fuzzing performance.

On the other hand, static rewriting is the opposite approach: instrument the target binary ahead of time, using various techniques to recover as much information as possible from the code (including heuristics). Flexibility of static rewriters is more limited, as virtually none of them support edge cases such as self-modifying binaries, code that does not support standard control flow practices (like OCaml or Haskell binaries), or even code generated by a non-popular compiler. However, the overhead introduced by static rewriting has generally low overhead, and moreover it can support more complex instrumentation passes (dynamic rewriters need to keep it simple for performance constraints).

Since static rewriters have much less flexibility, the aim of this thesis is to expand the domain of supported binaries of one of the recently published static rewriters, Retrowrite, by Dinesh et al. [9]. Retrowrite uses the relatively new approach of *symbolization*, that produces reassemblable assembly on which instrumentation passes may be applied to. Retrowrite focuses on position-independent binaries to eliminate the problem of distinguishing between scalars and pointers, and manages to produce reassemblable assembly without the usage of heuristics.

Retrowrite supports only x86_64 binaries in both user and kernel space; however, most mobile devices are running on an ARM CPU, and also some new desktop devices like the latest MacBook are switching to ARM. In this thesis we will explain our efforts to add support for userland ARM64 binaries, with the goal that one day we will be able to fuzz Android modules and closed source components. Furthermore, we will also explain how we ported to ARM the original instrumentation pass that the authors of Retrowrite implemented, *AddressSanitizer*, as a proof of concept to show the usefulness of the symbolization approach. To the best of our knowledge, there were no previous attempts at statically introducing memory sanitization to ARM binaries in the past.

We will show through benchmarks how the overhead of Retrowrite's instrumentation is very low when compared to source based instrumentation, and much lower than state-of-the-art dynamic instrumentation. In our experiments, we find that we are TODO: xx% slower than source based instrumentation, and TODO: xx% faster than state-of-the-art dynamic instrumentation.

Our core contributions consist in solving many challenges that static rewriting ARM binaries present, mostly spawning from the fact that ARM is a fixed size instruction set, in which pointers

need to be built in multiple instructions, or located in memory in a special region called a literal pool. A series of new problems like jump table size fixing arise from this, and in fact the number of static rewriters for ARM binaries is very limited, and, to the best of our knowledge, some of those problems were never analyzed before. We present our approach to fix or workaround most of those problems, and will give pointers to a few alternative solutions we analyzed.

We then evaluate our ARM port of Retrowrite , discuss its limitations, discuss related work, and present our conclusions.

Chapter 2

Background

Here we will provide a summary of the basic notions required to understand the underlying concepts behind Retrowrite and the problems we faced while porting it to the ARM architecture.

2.1 Binary Rewriting

Binary rewriting describes the alteration of a compiled program without having the source code at hand in such a way that the binary under investigation stays executable. The applications of binary rewriting are multiple and can be summarized as follows:

- **Emulation**: An emulator is a special software that mimics the behaviour of a device while executing on a different device. Emulators use binary rewriting to translate system calls, instructions, memory access and all the other execution primitives from one processor architecture to another. An example of this would be QEMU [3]
- **Optimization**: In high performance computing domains, having a way to patch subtle things like cache misses or timing anomalies in very long running tasks without the need to restart the whole program is of special interest. In such situations, binary rewriting is a solution for run-time patching, as it is done by DynInst [4] or Frida [1]

[1] `https://frida.re/`

- **Observation**: Having an in-depth look during the execution of a binary by inserting profiling or tracing instructions in the middle of its code can prove to be particularly useful in many applications, like catching memory leaks (e.g., Valgrind [15]), coverage information for fuzzing (e.g., AFL-QEMU [21]) and more.
- **Hardening**: The absence of source code, combined with no vendor support, or deprecated build tools, could make the use of binary rewriting necessary to introduce security measures such as the insertion of stack canaries (e.g., Stackguard [6]) or memory sanitization (e.g., Retrowrite [9]).

Instrumenting a binary is not an easy task, as it is often said that binaries have a *rigid* structure, because moving any data or code, or inserting instructions, will break all the references inside said binary. Thus, a binary rewriter must make sure that all the pointers and control flow broken by the inserted instrumentation must be corrected.

In the next section we will analyze the difference between the two different approaches that can be taken for binary rewriting, namely *dynamic* and *static* instrumentation.

2.2 Dynamic and Static Instrumentation

2.2.1 Dynamic instrumentation

Dynamic rewriters modify and instrument the code of the target binary during runtime. Usually, the target binary is executed in a controlled environment side by side with the rewriter engine, which patches instructions and fixes references on the go. Sometimes the rewriter engine leverages the operating system's primitives to control the execution of the target, like using the `ptrace` system call on Linux, but there are notable cases in which the rewriter engine comes with their own instrumentation runtime (e.g., Dynamo [1]) or implement a full featured virtual machine (e.g., STRATA [16]).

The big advantage of dynamic rewriters is the additional information that is only available at run time, like the path that the execution has taken. Furthermore, dynamic rewriters can avoid

analyzing the whole binary at once, as they can just focus on the part that is being currently executed, making them scalable to arbitrarily large programs.

However, the additional runtime information comes at a high performance cost: running the rewriter engine alongside the target binary is expensive, and furthermore the frequent context switches the CPU must do to execute both make the situation even worse. The total overhead for an instrumentation pass like memory sanitization for a state-of-the-art dynamic rewriter like Valgrind are in the order of 10x [9] the overhead introduced by source-level memory sanitization .

2.2.2 Static instrumentation

Static rewriters process the target binary *before* execution, and produce as output a new binary with all the required instrumentation already there. This approach nets a big advantage in the fact that, except in some rare cases, very low overhead is introduced, and execution speed comparable to compile-time instrumentation. Furthermore, static rewriters are able to add complex instrumentation that is computationally expensive to introduce, as since it is done statically, it won't introduce unnecessary overhead in the target binary.

Without runtime information, static rewriters need to rely on complex analysis to instrument, which are inherently imprecise, and sometime fall back on heuristics. It turns out that the common disadvantage of static rewriters is that they do not scale well on bigger binaries, or binaries that do not follow standard patterns. In fact, virtually no static rewriter supports packed binaries or self-modifying code. Moreover, many rewriters struggle with binaries produced by deprecated compilers or with aggressive optimization flags.

More recent static rewriters such as `Ramblr` [18], `Uroboros` [19], and `Retrowrite` rely on *symbolization* (sometimes also called *reassemblable assembly*), which work around the rigid structure of binaries by substituting hard coded references with assembly labels, that are more flexible and can be parsed by any generic off-the-shelf assembler. `Retrowrite`'s approach is particularly interesting in the fact that it avoid heuristics to differentiate between scalars and references by focusing on position-independent executables (PIE).

2.3 Instrumentation details

In this section we will go into more detail on our particular use-case of instrumentation, *fuzzing*, and explain what is the basis of AdressSanitizer, the instrumentation pass we implemented in the ARM port of Retrowrite.

2.3.1 Fuzzing

Automatic vulnerability discovery techniques are getting a lot of traction lately, mostly because software is getting ever more complex and large, and manual analysis and auditing does not scale very well. Fuzzing is certainly one of the most interesting automatic vulnerability discovery techniques. It relies on the pseudo-random generation of test cases to give as input to a target binary, trying to find a specific input that makes the binary get into an invalid or undefined state, as it is a good indicator of a possibly exploitable vulnerability. This technique got even more popular after the release of `AFL` [21], a fuzzer that relies on coverage information to generate new test cases to maximize the amount of instructions tested by each new input.

Most state-of-the-art fuzzers rely on instrumentation to improve vulnerability discovery, as it's a much more efficient to do so. When no instrumentation is available on a target binary, we usually use the term *Black-box fuzzing*.

2.3.2 ASan

AddressSanitizer, or ASan in short, is one of the most common static memory sanitization checks that can be added to a binary through a compiler pass, implemented in both the `clang` and `gcc` family of compilers. This compiler pass helps finding bugs by actively checking for memory corruptions, hooking calls to libc's `free` and `malloc` functions. ASan is not only used by developers to debug their code before releasing to production, but it is also extensively used by fuzzers, as ASan will detect a memory violation as soon as it happens, letting the fuzzer know faster and with more reliability when a bug was found.

ASan works by introducing a new region of memory called *shadow memory*, which a size of exactly 1/8 of the whole virtual memory available to a process. By keeping track of each call to `malloc` and `free`, ASan stores in the shadow memory a compressed layout of the valid memory in the original virtual space, and sets up *red zones* to highlight invalid or freed memory, that trigger a security violation as soon as they are accessed. Despite its non-negligible overhead (Around 73% on average [17]), ASan is widely used thanks to the absence of false-positives, and for its usefulness in detecting memory corruption vulnerabilities which are still extremely popular in C/C++ codebases.

2.4 The ARM architecture

We will provide a short summary of what are the main differences between x86 and ARM that proved to be source of non trivial problems during the porting of Retrowrite.

- *Fixed-size instruction set*: Contrary to x86, the ARM instructions are all of the same size, fixed to the value of 4 bytes. While this is not a problem in itself, it means that a pointer cannot fit into a single instruction. If you want to store a pointer in a register in ARM, there are two main options: the first is using a region of data where the pointer is hard-coded at compile time, called a *literal pool*; the second one is building the pointer in a multi-stage fashion by using arithmetic operations. While the first one is easier, it is also less performant, and compilers will always resort to the second when possible. This makes very hard recovering information about global variable accesses and callbacks.
- *Jump table compression*: On x86, jump tables are stored as list of pointers in a data section (usually `.rodata`), with one pointer for each case of the jump table. Instead, on ARM, jump tables are stored as *offsets from the base case*. This is because the compiler will try to compress the jump table, and in most cases a single byte is enough to indicate the offset from the base case to all other cases. This leads to problems for static rewriting because first of all jump tables are harder to detect, as on x86 scanning the data sections for arrays of valid instruction pointers was a quite reliable way of detecting jump tables, while on ARM they are indistinguishable from random bytes; secondly inserting too much

instrumentation between cases of the same jump table could lead to the offset not fitting into a single byte anymore, and breaking the whole jump table.

- *Discontinuities in immediates*: Some ARM instructions, like `ADD`, support having immediates as one of the operands. However, they do not accept a classical range of immediates like in x86, but instead a specific set of values that may not be continuous. For example the `ADD` instruction can use only immediates that can be expressed with a value of 8 bits scaled by a `ROR` with a 4 bits value.
- *Not enough mature tools*: The popularity of ARM CPUs is still relatively new and the tooling is not mature enough, as in fact we found bugs in both the disassembler we chose to use (Capstone [5]) and the debugger (GDB)

Chapter 3

Design

TODO:

3.1 Goals

TODO: Notes from Mathias: Tell me the design goals here! :)

TODO: Give a rundown of specific goals that we set ourselves at the start of the project:

- Study the ARM architecture and the main differences from x86
- Study the codebase of the Retrowrite project, understand the theory behind it and its implementation
- Write tests to ensure the robustness of the implementation
- Implement and introduce support for ARM in the Retrowrite project
- Develop a sample instrumentation pass (BASAN)
- Evaluate the resulting implementation against state-of-the-art techniques

3.2 Key Issues

TODO: Introduce and explain what will be the main problems we are going to face here:

- Alignment issues
- Global variables (!)
- Jump tables (!!)
- Control flow broken by too much instrumentation (e.g. short jumps)
- Literal pools (maybe move to only chapter 4, implementation?)

3.3 Symbolizer

TODO: Add a figure for the system overview here, with a short explanation on how it is split into a single Symbolizer and multiple possible instrumentation modules

TODO: High level rundown of how the original x86 Symbolizer works. Explain how we approached the symbolizer key issues mentioned above here.

3.4 Instrumentation (ASan)

TODO: Explain the ASan algorithm, the concept of shadow memory, etc. Explain the limitations of our binary ASan compared to source ASan:

- No checks on global variables
- Checks on the stack only at the stack-frame level
- Many more instructions instrumented because of lack of source code

Chapter 4

Implementation

4.1 Symbolizer

TODO: Go into the nitty-gritty details of my horrible hackish implementation,

4.1.1 Global variables

TODO:

How were they detected

How were they fixed

The literal pool dilemma

4.1.2 Jump Tables

How were they detected

TODO: Pseudo-emulating ARM binaries to detect jump tables

How were they fixed

TODO: Explain the following:

- List all ideas we had
- Explain why we chose to implement the ungodly hack of expanding them
- also, nop padding between cases

4.2 ASan instrumentation

TODO: Overview of the ASan algorithm, how it was hand-translated to ARM assembly TODO: Various optimizations that were introduced to lower instrumentation overhead

TODO:

Chapter 5

Evaluation

5.1 Experiment overview

TODO: Explain our setup, the hardware we used, what we were measuring.

TODO: Talk a bit on also the difficulties caused by the nature of the benchmarks, namely very long times (up to 48 hours, and probably more now that I have CloudLab :D), very high RAM usage, and so on.

TODO: Talk a bit on how those difficulties were approached, with the very strict collection of experiment metadata using Github CI and the telegram bot.

5.2 SPEC CPU

TODO: Explain how much do I hate the SPEC CPU benchmark, who developed it, the crazy fact that it is the accepted state of the art method for benchmarking single-core performance, how counter-intuitive their directory structure, config files, and bash scripts are. /s

5.3 Results

TODO: Present plots and tables, and a detailed explanation on each one. Make sure to respect the style of Mathias' previous papers/theses, to avoid getting him unnecessarily angry

TODO: Compare it to state-of-the-art dynamic instrumentation (QEMU).

Chapter 6

Related Work

TODO: This section now has only list of related work I would like to talk about, will be expanded later

Dynamic rewriters in general:
First, the good ol' ones: `PIN`[14], `Valgrind`[15] and `DynInst`[4].

More recently, `Multiverse` [2], `Frida` and `DynamorIO` are interesting examples.

`QASAN`[11] is similar to Retrowrite's BAsan, much slower but also much more portable (works on any binary supported by QEMU)

Static rewriters that use lifting to IR:
`McSema` [8] is a great example of LLVM IR lifting approach, also particularly nice as it supports C++ exceptions

Static rewriters that use trampolines:
`E9patch`[10] uses only trampolines, no need to recover control flow, but also noticeable overhead

Static rewriters that use symbolization:
`Uroboros`[19] and `Ramblr`[18] are the first reassemblable assembly approaches (symbolization), `Retrowrite` [9] (x86 version)

Static rewriters aimed at ARM binaries:

`Repica` [13] is probably the most recent addition to binary rewriters aimed at ARM binaries.

TODO: Maybe add Bistro ? [7]

For more check out recent surveys on the area of binary rewriting [20]

Chapter 7

Future Work

Support for more source languages: For now, `Retrowrite` supports only binaries compiled from the C language, both for the x86 and the ARM implementation. The easiest addition would be to add support for C++ by expanding the analysis capabilities of `Retrowrite` to support exception tables too, but many more languages could be supported in the future.

Support for kernel space binaries: Right now the ARM port of `Retrowrite` supports only userspace binaries, contrary to the x86 version that supports linux kernel modules too. The kernel version of `Retrowrite_ARM` would prove to be particularly interesting as it would open new ways to efficiently fuzz Android kernel modules.

Support for more executable formats/operating systems: The current implementation of the `Retrowrite` tool is aimed only towards ELF files, but adding support for MACH-O and PE binaries should not require too much effort. This would also be interesting as Windows and MacOS present way more closed-source modules compared to Linux.

More instrumentation passes: While right now we implemented only the AddressSanitizer instrumentation in the ARM port of `Retrowrite`, the design of `Retrowrite` is modular to make adding new instrumentation passes or new mitigations very easy for anyone. To name a few, the interesting ones would be:

- Shadow stack (return address protection)

- ARM pointer authentication (hardware-assisted)
- Control flow authentication
- Coverage-guidance for fuzzing

Chapter 8

Conclusion

In summary, we develop the ARM architecture implementation of the `Retrowrite` project, a scalable static rewriter for linux C binaries. `Retrowrite` enables targeted application of static instrumentation where no source code is available, such as proprietary binaries, inline assembly, or code generated by a deprecated compiler. We also present an example instrumentation pass on top of the symbolization engine, AddressSanitizer, particularly useful for fuzzing purposes. We present new solutions to problems arising from the peculiarities of the ARM architecture such as the fixed-size instruction set, global variable accesses and jump table instrumentation. We show that the total overhead of the symbolization and the instrumentation pass are competitive with source based AddressSanitizer. Our work shows that `Retrowrite`'s original approach is not limited to the x86 architecture, but can be applied to the ARM architecture and more.

Bibliography

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Transparent dynamic optimization: The design and implementation of Dynamo”. In: (1999).

[2] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. “Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics”. In: *Proceedings 2018 Network and Distributed System Security Symposium* (2018). DOI: 10.14722/ndss.2018.23300.

[3] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[4] Bryan Buck and Jeffrey K. Hollingsworth. “An API for Runtime Code Patching”. In: *The International Journal of High Performance Computing Applications* 14.4 (2000), pp. 317–329. DOI: 10.1177/109434200001400404.

[5] *Capstone: The Ultimate Disassembler*. http://www.capstone\protect\discretionary{\char\hyphenchar\font}{}{}engine.org. Accessed: 2020-11-02.

[6] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[7] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. “Bistro: Binary component extraction and embedding for software security applications”. In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 200–218.

[8] Artem Dinaburg and Andrew Ruef. “Mcsema: Static translation of x86 instructions to llvm”. In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

[9] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *IEEE International Symposium on Security and Privacy*. 2020.

[10] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. “Binary rewriting without control flow recovery.” In: *PLDI*. 2020, pp. 151–163.

[11] Andrea Fioraldi, Daniele Cono D’Elia, and Leonardo Querzoni. “Fuzzing binaries for memory safety errors with QASan”. In: *2020 IEEE Secure Development (SecDev)*. IEEE. 2020, pp. 23–30.

[12] *Global market share held by mobile operating systems*. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. Accessed: 2020-11-01.

[13] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. “REPICA: Rewriting Position Independent Code of ARM”. In: *IEEE Access* 6 (2018), pp. 50488–50509. DOI: 10.1109/access.2018.2868411.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[15] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[16] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. “Retargetable and reconfigurable software dynamic translation”. In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE. 2003, pp. 36–47.

[17] Konstantin Serebryany. *Hardware Memory Tagging to make C_C++ memory safe*. https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/Hardware%20Memory%20Tagging%20to%20make%20C_C++%20memory%20safe(r)%20-%20iSecCon%202018.pdf. Accessed: 2020-11-03.

[18] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. “Ramblr: Making Reassembly Great Again.” In: *NDSS*. 2017.

[19] Shuai Wang, Pei Wang, and Dinghao Wu. “Reassembleable disassembling”. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 627–642.

[20] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. “From hack to elaborate technique—a survey on binary rewriting”. In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–37.

[21] M. Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2020-11-03.