Introduction to Procedural Debugging through Binary Libification

Jonathan Brossard

Introduction to Procedural Debugging through Binary Libification

Jonathan Brossard, *Conservatoire National des Arts et Métiers, Paris*

https://www.usenix.org/conference/woot24/presentation/brossard

This paper is included in the Proceedings of the
18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Open access to the
Proceedings of the 18th USENIX WOOT
Conference on Offensive Technologies
is sponsored by USENIX.

Introduction to Procedural Debugging through Binary Libification

Jonathan Brossard
Conservatoire National des Arts et Métiers, Paris

Abstract

Assessing the existence, exact impact and exploitability of a known (or theoretical) memory corruption vulnerability in an arbitrary piece of compiled software has arguably not become simpler. The current methodology essentially boils down to writing an exploit - or at least a trigger - for each potential vulnerability. Writing an exploit for a weird machine involves several undecidable steps, starting with overcoming the reachability problem. In this article, we introduce the notions of "libification" and "procedural debugging" to facilitate partial debugging of binaries at the procedural level. These techniques allow the transformation of arbitrary dynamically linked ELF binaries into shared libraries, and the study of memory corruption bugs by directly calling the vulnerable functions, hence separating the memory corruption intraprocedural analysis from the reachability problem. Finally, we publish a framework [3] to implement such a libification under a permissive open-source license to facilitate its adoption within the security community.

1 Introduction

Triaging bugs has become an essential part of security. The Product Security function as a whole is becoming ever more critical for software manufacturers as legal frameworks around the globe mandate more clarity, speed, and transparency in dealing with existing and new vulnerabilities. The Cyber Resilience Act being implemented in Europe and the Executive Order on Improving the Nation's Cybersecurity published in the US, for instance, both mandate the use of Software Bill of Materials (sBOMs) and their communication to clients and third parties, effectively rendering the superficial - software version based - vulnerability assessment of potential new CVEs affecting their software, seemingly more apparent.

However, assessing the actual existence, exact impact, and exploitability of a given memory corruption bug, as required by the above laws, has not become significantly more manageable over time. The current methodology to assess the presence and impact of a given CVE in a piece of software essentially requires writing an exploit for each potential vulnerability. As such, this situation creates a seemingly unreasonable burden on Product Security teams, where triaging bugs requires performing operations like overcoming the reachability problem multiple times.

Writing exploits for a weird machine involves three steps: reaching, triggering, and exploiting. Much work has been done in automating the first step. Arguably, all of the fuzzing and dynamic testing performed hitherto follows this top-bottom approach, where execution starts from an application's entry point, toward the leaves of the application, across the application's call graph.

In this article, we aim to focus on the second step alone - without requiring solving the first one, which is undecidable in general.

Our methodology starts with modifying the ELF headers and dynamic section of an arbitrary dynamically linked ELF executable to turn it into a more workable shared library. The benefit of this technique is that any public function within the binary becomes callable without crafting an input to reach the attractive, potentially vulnerable function. Subsequently, we can render an arbitrary function within an ELF callable, even turn the entire ELF application into a callable API, and finally manually produce more limited, partial vulnerability triggers under the form of simple text files.

In the rest of this article, we will focus on memory corruption vulnerabilities unless stated otherwise and limit ourselves to C/C++ applications compiled as ELF binaries, as used under GNU/Linux and Unix-like operating systems, when implementing our framework. We will assume that the application's source code is unavailable to the auditor.

Our first contribution is a methodology to transform an arbitrary ET_EXEC or ET_DYN dynamically linked ELF binary into a shared library. We provide a tool named the Witchcraft Linker to perform this operation on ELF32 and ELF64 executables alike, regardless of their architectures. Our second contribution is a methodology to invoke arbitrary C or C++ functions within ELF shared libraries without prior knowledge of their exact prototypes. We implement an original type of debugger, procedural-based, allowing the invocation of arbitrary C/C++ functions. This debugger, the Witchcraft Shell, noticeably does not use ptrace(), breakpoints, or single stepping. We name this new form of debugging procedural since analysis is performed at the granularity of function calls.

2 Previous Work

2.1 Exploitability of a Vulnerability

As noted by Wang et al. [49], in general, the only definitive way to prove the exploitability of a vulnerability is to write an exploit for that vulnerability. This constitutes proof by construction since the expert exhibits an exploit that demonstrates the exploitability of the vulnerability. On the other hand, proving that a vulnerability is not exploitable is a difficult problem, according to Suciu et al. [44]. Demonstrating the non-exploitability of a vulnerability via formal proof based on a crash analysis is sometimes possible despite the explosive nature of proofs based on symbolic execution [9] [49].

Green et al. [24] consider that when it comes to vulnerabilities such as memory corruptions, the fact that attacker controls the next instruction to be executed (the "Program Counter") is a strong indicator of a function's exploitability. However, the presence of countermeasures may not make this condition entirely sufficient [20].

2.2 Automatic Exploitation of Vulnerabilities Detected via Static Analysis

Several research projects focus on exploiting (or at least triggering) vulnerabilities detected using a preliminary static analysis to demonstrate that they are true positives. ExpRace [31], for example, focuses on a single class of vulnerabilities: race conditions in the Linux kernel. After having distinguished race conditions involving several variables (qualified as difficult) and race conditions involving a single variable in the kernel (qualified as easy), the authors propose a generic methodology for exploiting reusable single-variable race conditions on several cores, running under Intel processors, making it possible to trigger the previously identified vulnerability, taking advantage of the fact that an unprivileged process (or secondary thread) in user mode can significantly increase the race window using common system calls (mmap() and mprotect()) to trigger the synchronization of memory tables ("Lookaside Buffers translation") between the cores of the same Intel microprocessor. The tool is very specialized since it only addresses the problem of mono-variable race conditions in kernel mode under Linux.

The FUZE tool [51] aims at dynamically triggering, to prove their existence, "Use After Free" vulnerabilities in kernel mode under Linux. By combining open-source frameworks such as syzcaller (fuzzer), angr [46] (for binary analysis, function call graph generation, decompilation, and symbolic execution), and kernel mode debugging techniques (parsing the list of kernel modules, "LKM linked list"), it dramatically reduces the complexity of UAF vulnerability analysis by determining the few paths and system calls that can potentially modify a variable in kernel mode, then using combined fuzzing and symbolic execution techniques to generate user inputs capable of automatically triggering the vulnerability, and thus proving its existence.

The article "A Hybrid Interface Recovery Method for Android Kernel Fuzzing" [32] is also specialized. The problem raised by the authors is the addition of undocumented interfaces (system calls or ioctls) between user and kernel modes by mobile phone equipment manufacturers. These new interfaces are typically additions via proprietary kernel modules (the source code of which is unavailable, implying an analysis partially to be made in black box mode) to the Android kernel (which is based on Linux and is, therefore, open-source, auditable in white box mode). However, these interfaces are prime targets for privilege escalation attacks, where a program in unprivileged user mode will purposely call these extra interfaces to the privileged mode of the kernel to trigger vulnerabilities. Therefore, the analysis is gray, combining a white box analysis of the open-source Android part of the kernel and a black box analysis of the non-open-source, proprietary part added by the equipment manufacturer. The methodology followed is a taint analysis of proprietary modules, including type propagation, to find the prototypes of the interfaces introduced (whether they are new system calls in their own right or, more commonly, new valid ioctl calls on arbitrary device drivers). Once the prototypes of these interfaces have been determined, it becomes possible to use classic whitebox fuzzing tools, such as Syzcaller, by measuring the impact of calling these new system calls dynamically on the rest of the kernel (id est: by instrumenting only the open-source part of the kernel).

The PhD thesis "Finding race conditions in kernels: from fuzzing to symbolic execution" [52] proposes an original approach to the detection and exploitation of "time of check, time of use" (or TOCTOU) vulnerabilities, which are a subclass of race conditions, where a kernel resource is validated at time t, then read back and used at time t+1. The underlying fundamental issue is that this resource may have changed in the meantime, the Linux kernel being multi-tasking and concurrent, leading to false assumptions on the said resource core properties. It should be noted that several vulnerabilities of this type have been discovered on the Linux kernel

in recent years, hence the renewed interest in an automatic approach to the discovery and practical validation of this peculiar vulnerability subclass. The methodology followed is to modify the Linux kernel (using source patches) to instrument regions likely to contain TOCTOU vulnerabilities, selected by a preliminary static analysis, then use a fuzzer guided by symbolic execution toward these regions to be more purposefully scrutinized. This methodology is limited to TOCTOU vulnerabilities and does not apply to kernels whose code is unavailable.

Furthermore, the article "From source code to crash test cases through software testing automation" [16] offers a methodology for creating proof of exploits (id est: the automatic generation of user input triggering a vulnerability, previously identified in source code), by combining a preliminary static analysis (generation of the call graph of the application) of the application, a fuzzing engine to traverse this graph, and the use of a symbolic execution engine (named Triton) to guide the fuzzer toward the vulnerable function. Although the source code is essential to this methodology, it applies to several classes of vulnerabilities, giving it notable genericity.

2.3 Defense in Depth: Hardened Compilation Techniques

Countermeasures have been developed to prevent or limit the exploitability of vulnerabilities in compiled applications, particularly those developed in C or C++. Detecting and taking into account, where applicable, the presence of these countermeasures is critical when writing an exploit taking advantage of memory corruption.

Khalsan et al. [28] identify in particular the DEP ("Data Execution Prevention") technique introduced in Microsoft Windows XP SP2, which makes the stack, dynamic memory, and variables in the data sections of an application non-executable. According to the authors, the non-execution of the stack is made possible thanks to hardware extensions ("NX" bit on AMD processors or "XD" equivalent on Intel processors). These countermeasures primarily aim to prevent the introduction and execution of shellcode [13] in all writable sections of the application. We also find the term W^X to name the segregation of variables (writable, non-executable) and code (executable, non-writable) in the literature [34] [10].

Khalsan et al. also describe the use of ASCII armoring, which ensures that all virtual addresses used by an application contain at least one 0x00 ASCII byte (in hexadecimal code). Given that the functions manipulating character strings end with a 0x00 (named ASCIIZ format), exploitating a stack buffer overflow vulnerability via the functions from libc making a copy of strings of characters is made impossible. Introducing ASCII armoring requires modifying the kernel and dynamic linker to provide armoring on the main binary and all its dynamic libraries.

Khasan et al. detail the use of Address Space Layout Randomization (ASLR) [43], which consists of making the base address of a binary and each dynamic library in memory non-predictable. An attacker can no longer hardcode return addresses when writing an exploit. The introduction of ASLR typically requires modifying the kernel and the file format of executables to allow arbitrary relocation of protected binaries [34].

Khasan et al. also describe binary protection techniques using canaries. These techniques have known several names, such as Propolice [21], StackGuard [15], and Stack Smashing Protection (SSP) [39]. This involves modifying the compiler in such a way as to introduce a canary (or "stack cookie") before the return address in the stack, the integrity of which will be checked in the prologue of each instrumented function. If the canary has been modified, the stack is corrupted, and the program will be immediately terminated rather than risk arbitrary code execution by an attacker. These techniques have undergone several successive improvements until they no longer have any significant cost during the execution of the protected application [53].

Khasan et al. finally detail FORTIFY (standardized in the ISO/IEC TR 247315 standard). This compilation option automatically replaces specific C library functions vulnerable to buffer overflows with functions including an additional argument, the maximum size of the destination buffer (which can often be inferred by the compiler). In the event of a stack buffer overflow during the program execution, the application is terminated rather than allowing the attacker to execute arbitrary code [30] [23].

These techniques have been extended to other architectures and operating systems, such as Linux [39], Android [33], OSX [39] or iOS [28].

Finally, there are protections against memory corruption at the hardware level of specific microprocessors, such as Intel Control Flow Integrity (CFI) [7] [29], which allows, by instrumenting the start of each block of code (an endbr64 instruction is added at compile time under Intel x86-64) [29] to ensure that the control flow of the application has not been altered via memory corruption exploitation techniques such as ret2libc [10] or Return Oriented Programming (ROP) [40] [34] [1] [12] at any point in time. During a transfer of execution via branching or when returning to a calling function, the microprocessor can ensure whether the destination address is an endbr64 instruction under x86-64 (respectively endbr32 under x86) and terminate the application if this is not the case.

These countermeasures to exploiting memory corruption vulnerabilities are effective against their respective vulnerability subclasses but require activation (often at compile time) to operate correctly [50].

2.4 Binary Loaders and Binary Post-Compilation Instrumentation

The idea of statically or dynamically loading and instrumenting binaries is fundamental in analyzing compiled applications.

The most basic form of dynamic instrumentation is simply using the trap instructions to force an application to divert its execution flow, as seen in DTrace [14].

A more complex tool like Valgrind and its popular memcheck [42] memory sanitizer can perform a Just in Time (JIT) dynamic recompilation of executables. It is a complex framework that starts by transforming the original basic blocks of the application into an intermediate representation, then applies instrumentation and code optimization before translating the intermediate representation back to machine code [36]. Such an instrumentation is heavy and incurs an execution penalty of 10x or higher.

Some of the techniques available include dynamically rewriting a single basic block of code at a time, while the application is running, using a shadow memory mechanism. This is the foundation of tools like DynamoRIO [4] [6] [5], a framework reused in popular security tools such as WinAFL [55].

Dinesh et al., on the other hand, opt for a pure static rewriting of binaries to retrofit into binaries instrumentation that is usually introduced at compile time, such as AFL [54] and Address Sanitizer [41]. Their framework, named RETROWRITE [17], works by diverting the flow of execution through the insertion of trampolines. A preliminary static analysis involves building the control flow graph, which is a difficult problem in general [35] and undecidable [22].

This mechanism, where a preliminary disassembly and control flow recovery precedes a static rewriting of portions of the binary to introduce instrumentation code, is a popular design [2] [48] [37] [47], subject essentially to the same limitations: recovery of the control flow is undecidable in general [22].

To avoid this pitfall, Duck et al. [19] developed a suite of binary rewriting techniques, implemented under the E9Patch framework, that can insert jump trampolines without requiring an understanding of the binary's control flow. As such, their instrumentation is more robust and scales to large applications such as web browsers. They leverage techniques such as instruction punning [11], which may safely replace branching conditions and introduce trampoline code by overwriting exactly one assembly instruction.

Furthermore, it is worth mentioning the idea of recovering individual object files from a compiled binary [8] thanks to a control flow and data flow analysis. When individual compilation units can be unlinked, they may be subsequently relinked and instrumented.

Finally, custom loaders may allow the loading of Windows dynamic libraries under Linux [38] or rewriting Windows Executables so they may be loaded as DLLs [18].

In light of this state of the art, it seems relevant to introduce a more lightweight form of binary rewriting focused solely on modifying an application's metadata. As such, it shall not suffer from the limitations of the techniques based on control flow recovery or the runtime penalty of dynamic instrumentation.

3 Overview of the Libification Process

3.1 Libification: Methodology

In this section, we describe the production of a libifier, that is to say, a tool able to reliably and automatically transform an arbitrary ELF binary into a shared library. We detail this methodology, so it may be extended in the future, if necessary, to compensate for breaking changes in the GNU dynamic linker, or adapted to other toolchains.

The POSIX 2001 standard specifies the API of the dynamic linker, and in particular the dlopen() function, which allows loading an arbitrary shared library in memory:

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flags);
```

The filename parameter must point to the path to the library to be loaded on the file system.

The flags parameter controls the locality (local or global) of the symbols loaded in the address space, as well as the behavior of the dynamic linker. In particular, if the RTLD_LAZY bit is set, the dynamic linker performs lazy binding of symbols when necessary, as opposed to an immediate binding at load time if the RTLD_NOW bit is set, in which case the Global Offset Table may be safely remapped read-only.

In the remainder of this chapter, we will define a shared library as an ELF file that can be loaded in memory via dlopen().

A minimal oracle to determine whether the dynamic linker can load an ELF file can be summarized with the following code:

```
int main (void) {
  void *handle = 0;

  handle = dlopen("./test.so", RTLD_NOW);

  if (handle <= 0) {
      printf(" !! ERROR: %s\n", dlerror());
      exit(EXIT_FAILURE);
  }

  printf("Loading successful\n");
  return 0;
}
```

If successful, the return code from this oracle will be 0. It will be non-zero otherwise, and an error message stemming from the dynamic linker will indicate the cause of the memory loading error. Empirically, the work of the libifier will, therefore, be to modify the binary in a way that prevents the error returned by dlerror() from occurring. The developer of the libifier will then read the code of the dynamic linker, identify the cause of the error, and modify the libifier to patch the input binary to prevent this last error from occurring.

The goal - and hope - of the developer of the libifier is that through this iterative and empirical process, the shared library produced by the libifier will be able to pass all of the dlopen() parsing checks, and eventually be loaded in memory. There is no guarantee that such a libification will be or remain possible in the future or across an arbitrary corpus of executables since this libification is a reverse engineering technique and not a standardized feature of a dynamic linker guaranteed in any form or fashion.

3.2 Practical Libification

The operations performed by the Witchcraft Linker to libify an arbitrary ELF binary modify the ELF header, the dynamic section, and the GNU-specific symbols versioning section of an input executable.

First, within the ELF header, the libifier must ensure that the e_type field is set to ET_DYN since all shared libraries are of type ET_DYN.

Then, the dynamic section of the ELF must be parsed and possibly modified:

The DT_BIND_NOW shall be changed to DT_NULL if present in the .dynamic section.

The DT_FLAGS_1 flags present in the .dynamic section may need to be modified: the DF_1_PIE and DF_1_NOOPEN bits must be removed if set. This last flag prevents an object from being loaded via dlopen().

If the binary features constructors or destructors, those may not expect to be called from dlopen(). The Witchcraft Linker, therefore, features an optional command line argument to prevent constructors and destructors from being called. Within the .dynamic section, setting the values of DT_INIT_ARRAYSZ and DT_INIT_ARRAY to zero inhibits the instantiation of constructors, and setting the values of DT_FINI_ARRAYSZ and DT_FINI_ARRAY to zero inhibits the calls to destructors.

Finally, because the dynamic linker may refuse to load multiple versions of symbols if symbols versioning is in use within the libified binary, the Witchcraft Linker will simply zero out the entire section of type SHT_GNU_versym.

Currently, the Witchcraft Linker (wld) version v0.0.6 can libify all of the binaries of a standard GNU/Linux distribution such as Ubuntu 22.04 LTS, so that they may be loaded via the dlopen() function of the GNU dynamic linker version 2.35.

3.3 Toward Procedural Debugging

Once the principle of libifying an ELF has been acquired, writing a debugger capable of loading a libified executable in its own address space is straightforward: simply load the libified binary via the dlopen() function of the dynamic linker. It appears appealing to integrate an interpreter into our debugger to allow a developer to interact with the functions exposed by the libified binary. Due to its tiny size, the choice of interpreter fell on the Lua language [25] since a Lua interpreter, including all its dependencies, occupies less than 500 kilobytes of memory footprint.

We wish to make the entire API available in the address space available to the Lua interpreter once the libified binary is loaded in memory. This API is made up, on the one hand, of the functions exported directly by the libified binary but also of the APIs exported by all the dynamic libraries loaded in memory by the dynamic linker when loading the libified binary in memory via dlopen(). The case of functions declared static and hence not exported at compile time is left aside for now[1]. Obtaining these APIs can be done via the use of the dlinfo() function of the dynamic linker [27] [45].

By making the entire API available in memory exposed to the Lua interpreter, we simply make these APIs available to the developer. One of the advantages of this methodology is that a developer or security analyst may invoke any function loaded in the address space without worrying too much about the actual calling conventions or prototypes (number and type of arguments) of these functions. Additionally, they may do so without compilation from a Lua interpreter, which facilitates manual exploration of said APIs.

We name this technique, which allows invoking a single function at a time, “procedural debugging”.

3.4 An Empirical Assessment of the Side Effects of Libification

In this section, we address the question of the side effects introduced by the libification of a binary over its main security hardening properties.

We successively consider the following properties: the base address of the executable mapping (ASLR), the presence of stack cookies aimed at preventing buffer overflows, the stack’s executability, the presence of static relocations (RELRO), and the presence of Control Flow Integrity type protections (Intel FCF).

Libification of an ET_DYN binary does not modify its ASLR properties: the binary being initially mappable to an arbitrary address remains so. In the case of the libification of an ET_EXEC binary, which was initially only mappable to a fixed address, the ASLR is not modified either: the library

[1] Static functions whose addresses relative to the base address of libraries or executables are known thanks to a preliminary control flow analysis may be named and called within the debugger.

thus generated is only mappable at the same address. Loading happens as if the binary had been transformed into a library by prelinking to the same base address [26].

The stack executability of a library loaded via dlopen() is determined by the stack executability of our debugger since the latter loads the library in its own address space. This debugger property can be arbitrarily changed via the execstack[2] application.

Libification does not modify the presence of static relocations (binary or library with the BIND_NOW flag in their dynamic sections).

The presence of stack cookies protecting the stack is intrinsic to each function since it is implemented by instrumenting the prologue and epilogue of each function. Libification, therefore, does not modify this property of the functions present in the libified binary.

The presence of Intel Integrity Protection (Intel FCF) type protections is characterized by the presence of endbr64 instructions at the start of each basic block in each protected function. Libification does not modify this intrinsic property either.

Finally, this empirical study overall suggests that libifying an ELF binary into a shared library does not modify its fundamental security properties, particularly the countermeasures possibly introduced into the binary at compile time. In a nutshell, libification does not introduce notable security side effects from an exploitability standpoint.

3.5 Limits to Binary Libification

Libifying an ET_EXEC binary as a shared library generates a somewhat special shared library since it cannot be remapped to an arbitrary address. This induces a limit to our libifier. On the one hand, a libified library can generate a collision with the address space of a program trying to load it, as noted by beta testers[3]. On the other hand, it is not possible to load two libified ET_EXEC binaries initially provided with the same base address in our debugger.

3.6 Validation

The libification process and the WSH debugger were validated under GNU/Linux Ubuntu 22.04 equipped with an Intel 64-bit processor using the following binaries:

Software	Version	Status	Time
Google Chrome	114.0.5735.198	OK	< 0.01s
OpenSSH Server	8.9p1	OK	< 0.01s
Apache2	2.4.52	OK	< 0.01s
Nginx	1.18.0	OK	< 0.01s
GCC	11.4.0	OK	< 0.01s

[2] https://linux.die.net/man/8/execstack

[3] Thanks to Dan Kaminsky https://github.com/endrazine/wcc/issues/26

Furthermore, copying, libifing, and loading via dlopen() the 435 binaries in the default path of a default Ubuntu 22.04 AMD64 install took less than 3 seconds (in total) on a laptop featuring a core i-7 11850H CPU and 32Gb of RAM.

4 Conclusion and Future Work

In this article, we presented a methodology to transform a dynamically linked ELF binary into a shared library. We called this methodology “libification”.

We then introduced a very simple debugger able to load such a libified executable within its own address space, hence rendering nonstatic functions within the binary callable. We named this technique facilitating the invocation of arbitrary functions in isolation and out of context “procedural debugging”.

Thus, a security analyst seeking to experiment with a possible vulnerability within an executable manually may now directly invoke the function featuring the vulnerability via procedural debugging without needing to produce user inputs traversing the application’s call graph before reaching the vulnerable function. This is notable since the reachability problem is undecidable in general.

We verified the reproducibility of the libification process on some of the most complex user-mode binaries available under GNU/Linux, as well as across an entire widespread Linux distribution, which validates the generality of the approach.

In the future, we hope to be able to automatically generate scripts to trigger a vulnerability within a compiled binary, which would save significant time for Product Security teams.

Availability

The Witchcraft Compiler Collection [3], including the Witchcraft Linker described in this article, is published under a permissive dual MIT/BSD open-source license. The framework is available from https://github.com/endrazine/wcc and via the package managers of several GNU/Linux distributions, including at least Debian, Ubuntu, and Arch Linux.

References

[1] Salman Ahmed, Long Cheng, Hans Liljestrand, N Asokan, and Danfeng Daphne Yao. Tutorial: Investigating advanced exploits for system security assurance. In *2021 IEEE Secure Development Conference (SecDev)*, pages 3–4. IEEE, 2021.

[2] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[3] Jonathan Brossard. The witchcraft compiler collection. https://zenodo.org/doi/10.5281/zenodo.11298208, May 2024.

[4] Derek Bruening and Saman Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. 2004.

[5] Derek Bruening and Timothy Garnett. Building dynamic instrumentation tools with dynamorio. In *Proc. Int. Conf. IEEE/ACM Code Generation and Optimi zation (CGO), Shen Zhen, China*, 2013.

[6] Derek Bruening and Qin Zhao. Building dynamic instrumentation tools with dynamorio.

[7] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[8] Mauro Capeletti. Unlinker: an approach to identify original compilation units in stripped binaries. 2016.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[10] S Sibi Chakkaravarthy, Dhamodara Sangeetha, and V Vaidehi. A survey on malware analysis and mitigation techniques. *Computer Science Review*, 32:1–23, 2019.

[11] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–332, 2017.

[12] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[13] Tsung-Huan Cheng, Ying-Dar Lin, Yuan-Cheng Lai, and Po-Ching Lin. Evasion techniques: Sneaking through your intrusion detection/prevention systems. *IEEE Communications Surveys & Tutorials*, 14(4):1011–1020, 2011.

[14] Greg Cooper. Dtrace: dynamic tracing in oracle solaris, mac os x, and free bsd by brendan gregg and jim mauro. *ACM SIGSOFT Software Engineering Notes*, 37:34, 2012.

[15] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, 1999.

[16] Robin David, Jonathan Salwan, and Justin Bourroux. From source code to crash test-cases through software testing automation. *Proceedings of the 28th C&ESAR*, page 27, 2021.

[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[18] Aleksandra Doniec. Converts a exe into dll. https://github.com/hasherezade/exe_to_dll, 2020.

[19] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 151–163, 2020.

[20] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017.

[21] Hiroaki Etoh and Kunikazu Yoda. Propolice: Protecting from stack-smashing attacks. *Technical Report, IBM Research Division, Tokyo Research Laboratory*, 2000.

[22] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[23] Jeff Gennari, Shaun Hedrick, Frederick W Long, Justin Pincar, and Robert C Seacord. Ranged integers for the c programming language. 2007.

[24] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Proceedings on Privacy Enhancing Technologies*, 2023.

[25] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[26] Changhee Jung, Duk-Kyun Woo, Kanghee Kim, and Sung-Soo Lim. Performance characterization of prelinking and preloadingfor embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 213–220, 2007.

[27] David Keller, Timothy Roscoe, Reto Achermann, and Simon Gerber. Bachelor's thesis nr. 137b.

[28] Mahmood Jasim Khalsan and Michael Opoku Agyeman. An overview of prevention/mitigation against memory corruption attack. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, pages 1–6, 2018.

[29] Sandeep Kumar, Diksha Moolchandani, and Smruti R Sarangi. Hardware-assisted mechanisms to enforce control flow integrity: A comprehensive survey. *Journal of Systems Architecture*, 130:102644, 2022.

[30] Marc-André Laverdière, Serguei A Mokhov, and Djamel Benredjem. On implementation of a safer c library, iso/iec tr 24731. *arXiv preprint arXiv:0906.2512*, 2009.

[31] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.

[32] Shuaibing Lu, Xiaohui Kuang, Yuanping Nie, and Zhechao Lin. A hybrid interface recovery method for android kernels fuzzing. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 335–346. IEEE, 2020.

[33] Héctor Marco-Gisbert and Ismael Ripoll-Ripoll. Sspfa: effective stack smashing protection for android os. *International Journal of Information Security*, 18(4):519–532, 2019.

[34] Jonathan AP Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749. IEEE, 2012.

[35] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[36] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.

[37] Trail of Bits. Mcsema. https://github.com/lifting-bits/mcsema, 2020.

[38] Tavis Ormandy. Porting windows dynamic link libraries to linux. https://github.com/taviso/loadlibrary, 2017.

[39] Conor Pirry, Hector Marco-Gisbert, and Carolyn Begg. A review of memory errors exploitation in x86-64. *Computers*, 9(2):48, 2020.

[40] Yefeng Ruan, Sivapriya Kalyanasundaram, and Xukai Zou. Survey of return-oriented programming defense mechanisms. *Security and Communication Networks*, 9(10):1247–1265, 2016.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[42] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[43] Zhidong Shen and Weiying Chen. A survey of research on runtime rerandomization under memory disclosure. *IEEE Access*, 7:105432–105440, 2019.

[44] Octavian Suciu, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitraș. Expected exploitability: Predicting the development of functional vulnerability exploits. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 377–394, 2022.

[45] Justin Tracey. Building a better tor experimentation platform from the magic of dynamic elfs. Master's thesis, University of Waterloo, 2017.

[46] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[47] 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.

[48] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[49] Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2(1):1–25, 2019.

[50] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. A survey of exploitation techniques and defenses for program data attacks. *Journal of Network and Computer Applications*, 154:102534, 2020.

[51] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[52] Meng Xu. Finding race conditions in kernels: The symbolic way and the fuzzy way. 2020.

[53] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438. IEEE, 2006.

[54] Michal Zalewski. Afl: American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2016.

[55] Google Project Zero. Winafl. https://github.com/googleprojectzero/winafl, 2016.