Sleak: Automating Address Space Layout Derandomization

Christophe Hauser
Information Sciences Institute
University of Southern California

Jayakrishna Menon
Information Sciences Institute
University of Southern California
/Arizona State University

Yan Shoshitaishvili
Arizona State University

Ruoyu Wang
Arizona State University

Giovanni Vigna
University of California, Santa Barbara

Christopher Kruegel
University of California, Santa Barbara

ABSTRACT

We present a novel approach to automatically recover information about the address space layout of remote processes in the presence of Address Space Layout Randomization (ASLR). Our system, dubbed *Sleak*, performs static analysis and symbolic execution of binary executable programs, and identifies program paths and input parameters leading to *partial* (*i.e.*, only a few bits) or *complete* (*i.e.,* the whole address) information disclosure vulnerabilities, revealing addresses of known objects of the target service or application. *Sleak* takes, as input, the binary executable program, and generates a *symbolic expression* for each program output that leaks information about the addresses of objects, such as stack variables, heap structures, or function pointers. By comparing these expressions with the concrete output of a remote process executing the same binary program image, our system is able to recover from a few bits to whole addresses of objects of the target application or service. Discovering the address of a single object in the target application is often enough to guess the layout of entire sections of the address space, which can be leveraged by attackers to bypass ASLR.

CCS CONCEPTS

• **Security and privacy** → *Logic and verification*; *Software reverse engineering*.

KEYWORDS

Binary program analysis, vulnerability discovery, information leakage

ACM Reference Format:
Christophe Hauser, Jayakrishna Menon, Yan Shoshitaishvili, Ruoyu Wang, Giovanni Vigna, and Christopher Kruegel. 2019. Sleak: Automating Address Space Layout Derandomization. In *2019 Annual Computer Security Applications Conference (ACSAC'19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359820

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359820

1 INTRODUCTION

In recent years, mitigation techniques against memory corruption vulnerabilities made their way into most of the major operating systems and compilers. Apart from application-level hardening mechanisms such as *stack canaries* [15], two major OS-level techniques were introduced: OpenBSD's $W \oplus X$ [3] (or equivalently, Microsoft's DEP) and ASLR from the PaX Team [4]. The former makes the stack non-executable, while the latter loads the code of external libraries to randomized locations (*i.e.,* randomized base addresses), and positions both the stack and the heap in random locations. While exploitable programming errors (in particular, buffer overflows) remain very common in commodity software, such mitigation techniques raised the bar to the next level when it comes to exploitation. Circumventing $W \oplus X$ or DEP requires code reuse attacks such as return to libc [43] or Return Oriented Programming (ROP) [35]. These attacks rely on the existence of so-called *code gadgets*, which are found in the code of shared libraries, and require knowledge of their addresses in memory. ASLR, in turn, randomizes such addresses, and makes it difficult, from the attacker's standpoint, to a-priori[1] know the location of code gadgets. As long as the address space layout of the targeted process remains *secret*, attackers will very unlikely be able mount successful ROP attacks against it.

It is worth mentioning that there are exceptions to this: both services restarting themselves after crashing, and programs running on 32-bit platforms, let the door open to statistical attacks [10, 36], allowing the attacker to recover the location of code gadgets. However, in the general case, ROP attacks require prior knowledge of the layout of the target's address space.

This forces attackers to craft more sophisticated attacks by employing a two-step approach. In the first step, the attacker tries to gather information about the address space layout in order to recover the location of code gadgets. This is typically done by identifying and exploiting a *memory disclosure* vulnerability in the targeted application [40]. In the second step, a ROP attack is mounted against the targeted memory corruption vulnerability.

The requirement of leaking process information for successful exploitation led to a gain in interest around the process of discovering information disclosure vulnerabilities, and modern exploits rely on a number of *info leaks* techniques [34]. These techniques exploit common logic bugs to leak information from memory, such as arbitrary reads caused by buffer over-reads, type confusion, or

[1] Without prior knowledge of the base address of the program, or any address such as a function pointer.

use after free, amongst others. Apart from leaking addresses, information disclosure vulnerabilities are also commonly exploited by attackers to leak sensitive data such as cryptographic keys or passwords, as in the famous case of the *Heartbleed* bug [17], for instance, which remained unnoticed for a long period of time before it was discovered.

Due to the complexity of the software involved, it is difficult for existing automated tools to detect deep bugs without additional knowledge of the specific target environment and specificities of the analyzed software, and, in practice, manual analysis along with fuzzing are used in most cases to discover new vulnerabilities [12].

However, the process of manual analysis is both non-exhaustive and error-prone, which results in a lot of vulnerabilities remaining (publicly) undiscovered. In addition to this, despite being a very active research topic, the coverage of fuzzing techniques remains limited (e.g., program branches depending on specific conditions such as a hash value are very unlikely explored), resulting in bugs being found in either shallow paths or specific program branches.

The situation is even more critical when it comes to binary software, for which the source code is not available, as it makes the manual analysis and reverse engineering processes considerably harder. Yet, proprietary software distributed in binary-only form is ubiquitous. An example of this is Adobe's flash player, which because of its popularity and error-ridden implementation is regularly subject to new discovered vulnerabilities, *e.g.,* `CVE-2018-4944`, `CVE-2018-4934` or `CVE-2018-4920`.

Existing approaches to detect information leakage either focus on out-of-bound reads, type confusion or use-after-free using dynamic approaches or require source code [16, 21, 24, 26, 33]. However, as of today, little attention has been spent in the research community to address the detection of *indirect* information leaks in binary software. In this paper, we introduce *Sleak*, a novel approach to detecting and exploiting information disclosure vulnerabilities in binary software, based on static program analysis and symbolic execution. This approach targets unsafe pointer usage and manipulation leading to partial or complete address leakage.

The underlying intuition behind our approach is that, by observing outputs of a program, an attacker can infer information about randomized addresses corresponding to its internal objects[2] (*e.g.,* stack variables), if there exists any dependence between such an object and any output of the program. A dependence of this kind may be introduced accidentally (*e.g.,* through a pointer manipulation error) or voluntarily (*e.g.,* the programmer intended to use the address as a unique identifier). Either way, those information leaks are, by nature, indirect, and involve a number of operations transforming the address into a value. Without prior knowledge of the aforementioned transformations, it is very difficult for an attacker to differentiate such an output value, leaking sensitive information about the address space of the program, from a benign value, as only part of an address may be revealed. In other terms, transformations over partial pointer addresses "look" just like any other random value from an observer's standpoint.

Our system addresses this problem by providing to the observer the ability to identify which bits of an output value correspond to transformations (or computations) over an address, or *which bits of the output indirectly leak address information.* Once such bits have been identified, our system builds *symbolic expressions* that represent the values of those bits in terms of addresses of the program. This accurate tracking of address information relies on symbolic execution of code at the binary level, by following paths from address definitions to the use of subsequent data as part of an output of the program. As a result, based on a symbolic expression and a concrete output value of a running instance of a program, our system is capable of reconstructing parts of the (indirectly) leaked address, up to the full address, depending on the nature of the transormations.

Our approach does not necessarily require the attacker to interact with the targeted remote process, and in that sense, can be done passively, as long as the attacker is able to *observe* its outputs (*e.g.,* network packets). It only requires prior knowledge of the binary code of the corresponding program. In order to attack a remote process, the attacker needs to i) run our analysis locally so as to identify address leaks, and to obtain symbolic expressions of the relevant output values of the program, and ii) solve symbolic constraints on these expressions given the knowledge of a concrete output value. The approach we propose is fine-grained and detects indirect information leaks at the bit level. Depending on the "size" of the leak, that is, the number of bits of address that are revealed in a given output, the attacker may directly and precisely reconstruct the original address, or may need to brute force a range of addresses containing the original address. In the latter case, leaking bits of an address has the potential to significantly reduce the entropy of ASLR, especially since most current implementations use limited address pools due to practical constraints (*e.g.,* user/kernel separation, stack located higher than heap, etc.).

Architecture	Stack	Heap	mmap
32-bit	19	13	8
64-bit	30	28	28

Figure 1: Bits of entropy per memory region (Linux 4.5.0)

Shacham *et al.* [36] have demonstrated that, in practice, a 16-bit entropy is not enough to prevent brute-force attacks against ASLR (the authors also estimated that 20-bit remains within the reach of practical attacks). As represented in Figure 1, current 32-bit Linux systems are exposed to derandomization attacks, while the entropy on current 64-bit Linux systems is significantly stronger. As these numbers [25] suggest, it is therefore necessary for an attacker to leak from 8 to 10 bits (roughly, a byte) of an address before considering a brute-force attack. However, the "effective" entropy may be reduced in practice, as shown by Herlands *et al.* [19] by leveraging lower-entropy regions[3], which, if not directly exploitable, requires the attacker to learn only a few bits of address.

To the best of our knowledge, this present work is the first approach to automatically identify memory disclosures at the bit level granularity in off-the-shelf binary applications. In summary, our contributions are the following:

[2]By objects, we refer to internal constructs such as stack variables, heap buffers or functions being accessed through their addresses.

[3]As of today, non-PIE (Position Independent Executables) as well as backward compatibility with 32-bit applications still expose 64-bit systems to low-entropy regions.

- We leverage binary program analysis techniques in a novel way in order to detect sensible code paths leaking address information.
- We design a fine-grained model to identify address definitions in binary code, and to track address dependency in the arguments of output functions that external attackers may be able to observe.
- We present a prototype and evaluate it on user-space applications, a commonly used general purpose library as well as in a filesystem implementation shipped with the Linux kernel.

2 APPROACH OVERVIEW

Let us consider a vulnerable server-side authentication function. It first receives an authentication token through the network, and verifies its validity. The data structure representing the token, as shown in Figure 2, embeds a union `auth` within a `token` C structure, containing two fields of different sizes. Accidentally accessing the union using the wrong field leads to a (partial) address leak since its members have different sizes. It should be emphasized that this type of construct is commonly encountered in libraries, such as libSDL's event handling code[4], for instance, and that such vulnerabilities are common. In fact, similar type confusion vulnerabilities have been discovered in the past[5] in Adobe Flash Player, in the PHP interpreter and in the Chrome browser, to mention only a few.

Analyzing code in binary form, as shown in Figure 3, is much less intuitive than the source code version of the same program, due to the lack of type information about the data structures. Figure 3 shows the relevant basic block involving the unsafe operation. In this basic block, the address of the `token` is first stored in the register `rax`, and then used to access the memory location of the `auth` union. The content of this memory location is then passed as a parameter to `sprintf` through the register `rdx`. It is important to note that, at the assembly level, no distinction is made between the members of the union as both correspond to the same underlying memory location. Another aspect to consider is that the generated output likely will not fall within the range of an address, since only part of the address is leaked: this bug causes the value of a `char *` pointer to be interpreted as an `unsigned int`. Such an address leak is not obvious when simply observing the output of the program.

```
typedef struct _token {
        int type;
        union{
        char *pass;
        unsigned int key;
        }auth;
        int status;
        // [...]
}token;
#define KEY(s)              s->auth.key
#define type_pass       0
#define type_key        1
#define valid               0xff
```

Figure 2: Definitions in header file.

[4]http://www.libsdl.org/release/SDL-1.2.15/docs/html/sdlevent.html
[5]CVE-2015-3077, CVE-2014-4721 and CVE-2015-1302.

```
mov    rax,QWORD PTR [rbp-0x18]
mov    rdx,DWORD PTR [rax+0x8]
mov    rax,QWORD PTR [rip+0x200516]
mov    esi,0x400754
mov    rdi,rax
mov    eax,0x0
call   4004b0 <sprintf@plt>
```

Figure 3: ASM representation

For additional details, the source code of this network authentication example and a more detailed description of the vulnerability are provided in Appendix 8.2.

2.1 Challenges

While it is possible from the source code version of this program to determine the types of variables and the layout of data structures in memory, this information is absent from its assembly translation. This has a direct impact on the complexity of the analysis, whether it is manual or automatic. Without further information about memory and register content, it is difficult to detect a vulnerability that leaks information.

In order to retrieve information from binary programs, our approach leverages symbolic execution, and accurately tracks expressions that depend on addresses. Symbolic execution allows us to keep tracks of variable expressions and constraints, and makes it possible to quantify and determine leaked information at the bit level. However, one of the intrinsic limitations of symbolic execution is the problem of exponential path explosion. Even when using techniques such as Veritesting and path prioritization [7, 31, 41], it is often infeasible to analyze large programs symbolically. As a result, avoiding path explosion while keeping an acceptable coverage of the program represents a challenge.

Our approach, in response, leverages a combination of static and symbolic program analysis techniques in a novel way in order to focus on analyzing the relevant paths of the analyzed binary program, *i.e.*,, where information about addresses may leak. More precisely, our approach involves the following analysis phases.

2.2 Analysis phases

Path selection and address identification. The first phase of our analysis automatically identifies code paths and locations of interest within the binary. During this phase, *Sleak* operates as follows:

(1) Control-flow recovery: Sleak starts by generating a control flow graph (CFG) of the analyzed binary program, in order to recover the location of output functions, which are later used as sinks in our analysis. During this step, a static control-flow graph is built, and program paths involving output functions are identified.

(2) Address identification: On each identified program path, *Sleak* identifies addresses by using a number of *inference rules* described in Section 3.3. This step identifies and marks the set of program locations involving addresses on each path.

Leak detection and address reconstruction. The second phase of our analysis leverages symbolic execution and constraint solving in order to accurately detect and determine *what* is leaked and to reconstruct address information from the leaked program output.

(3) Detecting address dependence: Sleak determines whether the arguments of output functions are data-dependent on marked addresses. The control-flow paths leading to each sink are analyzed individually. Each selected path is analyzed through a symbolic execution engine, which generates symbolic expressions and constraints on the program's variables as the path is executed. This step allows us to precisely characterize the leaked address, as a symbolic expression, or *formula*.

(4) De-randomization: Once *Sleak* reveals the set of leaking outputs for a given program along with their *formulas*, an attacker is able, *based on the observation of a single concrete output of the program corresponding to this variable*, to infer the values of the leaked bits of address by using a constraint solver, and therefore to de-randomize the base address of the corresponding object (main binary or library) within the address space of the remote process. This process is described in §4.7.

2.3 Automation, scope and objectives

Our approach to detect sensible code paths and to generate symbolic expressions of leaked addresses is entirely automated based on the knowledge of the binary executable image of the program to analyze. In order to de-randomize addresses, it requires, as input, the value of one instance of concrete output corresponding to one of the detected leaking paths. From this knowledge, our analysis returns a set of solution addresses corresponding to the leaked object.

The process of interacting with a remote service is outside of the scope of this work, and expected to be performed manually by a human operator. Similarly, while this approach may be leveraged in order to automate control-flow hijacking attacks (which typically require such an information leak in order to bypass ASLR), it is outside of the scope of this work. In this present work, we focus on defeating ASLR by exposing the symbolic expression and constraints over pointer addresses based on our automated binary-level approach.

The remainder of this paper presents our approach in more details. In Section 3, we present *Sleak*'s static analysis phase, which selects program paths of interest in a lightweight and scalable manner. Then, in Section 4, we introduce our symbolic execution model, along with our and address recovery mechanisms. We describe our evaluation on real-world software in Section 5, followed by a discussion of our approach in Section 6 and related work in Section 7. We finally conclude in Section 8.

3 PATH SELECTION AND ADDRESS IDENTIFICATION

As previously mentioned in Section 2, adopting a purely symbolic exploration of the target program is very likely to cause path explosion due to the large number of paths encountered by our analysis system. Furthermore a large amount of library code is involved in commodity software, which dramatically increases the amount of code to analyze. This involves analyzing complex code paths going back and forth between the main binary and the libraries. Code paths of interest involve data dependencies between an address (whether it is hardcoded or generated dynamically) and the argument of an output function. This section presents our approach to identify sensible code paths of interest on which to focus our analysis. The outcome of this is a set of program paths originating from *sources* and terminating in *sinks*.

3.1 Control-flow recovery

Sleak builds on top of standard techniques for static disassembly and control-flow recovery, as provided by [1], which it augments with novel insights and heuristics. The disassembled code is lifted to an Intermediate Representation (IR) as part of the disassembly process, and our analyses operate at this level of abstraction.

Sleak's first analysis stage consists in recovering an *interprocedural* static Control Flow Graph (CFG) of the binary, along with basic coarse information about the program state at the entry point of each node. This analysis step is to be thought of as a static pre-filter: for efficiency reasons, the recovered CFG in this step is *not context sensitive*, this allows us to scale our analysis to larger code bases, at the cost of a limited accuracy. A *context sensitive* control-flow recovery would consider multiple possible call site contexts into account during the analysis, *i.e.,* analyze each basic block considering the context of each potential *caller* when generating the CFG. However, such a context-sensitive analysis would also come with a considerable increase in terms of complexity. By omitting the context, *Sleak* reduces the complexity by an order of magnitude and allow our analysis to scale to larger binaries. On top of this context-insensitive CFG, *Sleak* performs lightweight data dependency tracking within the basic blocks surrounding output functions. In these blocks, constant values that are placed in the binary by the compiler (and loaded in registers or memory as immediate), as well as the result of trivial operations, are evaluated, and flagged as addresses, if they match one of the inference rules presented in paragraph 3.4. In the presence of such potential addresses, *Sleak* tracks dependency between registers and temporaries at the Intermediate Representation (IR) level and attempt to identify any data dependence[6] towards the arguments of one of the output functions.

In summary, this initial analysis step is used to build a CFG and to identify constants and simple cases of address-dependent outputs of the program at a small computational cost. Based on this information, *Sleak* proceeds with the identification of output functions, and their call sites within the binary image of the program and its libraries.

3.2 Output functions identification

When a program depends on the code of shared libraries, its binary translation is either statically or dynamically linked to the corresponding library code. In the case of dynamically-linked binaries, output functions are, in the vast majority of cases, external to the binary and imported as part of a library. It is extremely unlikely to encounter a binary reimplementing its own output functions such as `printf`, therefore we assume that all output functions are part of external shared libraries. The addresses of such library functions are exposed in the global offset table of dynamically-linked binaries, and functions are called through their respective procedure linkage table (PLT) entries. In this case, extracting function information is trivial. However, in the case of statically-linked binaries, function calls are performed directly (as opposed to calling a stub for resolution), and if a binary does not contain symbol information (*i.e.,* it

[6] That is, address information used as data.

was stripped), then no information about the location of functions is available. In this case, we perform a preliminary step of function identification, as described in Appendix 8.1.

In the reminder of this section, we assume that we are working with dynamically-linked binaries, and that program outputs rely on standard library functions. We also assume that we know the prototype of such standard functions (*i.e.,* we know the type of their parameters). These assumptions are reasonable, since in practice, the vast majority of programs rely on the C standard library, which provides system-call wrappers and the implementation of the most common input/output functions.

Once output functions have been identified, *Sleak* locates their call sites by iterating through each node of the CFG while scanning for the targets of `call` instructions. We refer to this analysis step as `SA1`. At the end of this step, all reachable, known output functions as well as their call sites have been identified. Each call site represents a potential address *sink* in the following analysis steps.

3.3 Identifying addresses

The next step through identifying address leakage is to identify memory locations containing addresses, which we treat as *sources*. In binary form, addresses can be encoded in different manners: when the address of a symbol is known at compilation time, the compiler will substitute the symbol with an immediate value, representing *e.g.* the address of a Procedure Linkage Table (PLT) stub, or a global variable. If the binary is compiled to be position independent (PIC), addresses will often[7] be encoded as offsets from the current value of the instruction pointer (*i.e.,* `$rip + offset` in x86_64 assembly). In other cases, addresses may be evaluated at runtime from simple (*e.g.*, offset from stack pointer) to complex expressions that are either difficult to determine statically, or not statically computable without specific knowledge about the application (*e.g.,* functions registered at runtime).

Sleak identifies addresses based on the set of *inference rules* described below in § 3.4. We refer to this analysis step as `SA2`.

3.4 Address inference

While destinations of jump targets can easily be flagged as being addresses, the distinction between addresses and data is not obvious in other situations. Consider the following assignment, for example, along with its assembly translation:

```
(C)    x = (char*) &printf;
(ASM)  mov QWORD PTR [rax],0x4003e0
```

While it is clear that `rax` corresponds to an address (as its contained value is dereferenced), `0x4003e0` is an immediate value, and it may never be used as an address in the program. We may also encounter cases where the actual value of the operand is unknown by our analysis. For example, consider the following statement:

```
mov rax, QWORD PTR [rdi+0x4]
```

Here, the value located in memory at `[rdi+0x4]` is read, and stored in the register `rax`. Without further analysis about the context, we cannot tell whether `[rdi+0x4]` corresponds to an address, or to some data. In order to cope with this lack of information, we use the following inference rules to determine whether a value potentially corresponds to a valid address:

(1) Semantic information: when analyzing dynamically-linked binaries, we can generally extract information from the binary format, such as the location of Global Offset Table (GOT), or relocation information[8]. The GOT contains addresses of external library functions that are called from within the binary, therefore we know that any memory read from these locations will contain an address.

(2) Value range: if a value falls into the boundaries of the `.text`, `.data`, `heap` or `stack` regions of the address space, then this value is a potential address and flagged for further analysis.

(3) IR operations: the set of operations available at the intermediate representation level expects values of different types. If a value is used as an address as part of an operation (*e.g.,* a memory `load`, `store` or a `jump` instruction), then it is flagged accordingly.

(4) Return values: the return values of libc functions or system calls, known to return pointers to memory locations, such as `malloc` or `mmap`, are tracked as part of our analysis.

3.5 Dynamic resolution

In some large software components such as operating system kernels or complex libraries implementing generic programmatic constructs (*e.g.,* parsers), dynamic behavior such as runtime binding, asynchronous method invocations and polymorphism tend to challenge static analysis, and it is sometimes necessary to provide hints to static analysis methods in order to resolve part of the control flow. For instance, filesystems in the linux kernel register a `inode_operations` structure when the initialization routines of the related modules are executed. Predicting such a behavior statically, without prior knowledge of the inner mechanisms of the kernel is not practical.

In order to reason about such programs exhibiting a highly dynamic behavior, we rely on *partial* concrete execution of the code in order to initialize the program to a reasonable state prior to proceeding with our analysis. In order to do so, we leverage dynamic testcases or known well-formed input to the program and collect execution traces. From these traces, we extract the execution context at the entry points of each encountered function, which we feed as initial state to our analysis. From there, our analysis proceeds with path selection and address identification in an identical manner to what is performed in a purely static setting.

In conclusion, the first phase of our analysis performs the recovery of a control flow graph of the binary program, and identifies sensible control-flow paths between potential sources and sinks. The next section describes the next phase of our analysis, based on symbolic execution.

4 LEAK DETECTION AND ADDRESS RECONSTRUCTION

Sleak leverages symbolic execution to accurately reason about address leaks. In this section, we describe why and how we use symbolic execution on vulnerable program paths (described in Section 3)

[7] Depending on the architecture support for instruction pointer addressing.

[8] Even when debug information is not present or the binary is stripped, the loader requires a certain amount of information concerning the location of certain GOT slots along with relocations to perform.

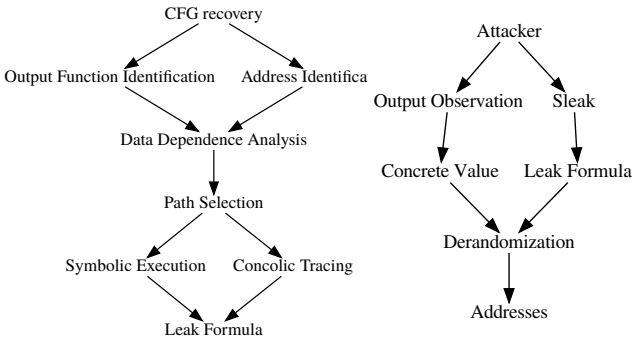

Figure 4: Leak detection and address reconstruction

and summarize the techniques involved. An overview is presented in Figure 4.

4.1 Identifying vulnerable paths

The key property of program paths leaking address information is that, within such paths, a data dependence exists in the program between a *source* and a *sink*. A source corresponds to an instruction i_1 where a *location* such as an instance of a register or a variable in memory) containing the value of an address is accessed. A *sink, i.e.,* another instruction i_2 where an argument is passed to an output function. In order to detect vulnerable paths, *Sleak* analyzes the existence of data dependence between sources and sinks which have been identified during the previous analysis phase presented in §3.

We now present *Sleak*'s data-dependence analysis in more detail. We first describe some limitations of static data-dependence tracking techniques at the binary-level, followed by our approach to overcome such limitations in the context of our analysis.

4.2 Limitations of static techniques

The generation of data dependence graphs is a common problem of data-flow analysis, and established algorithms exist in the compiler and source-level analysis literature [5, 42], to compute such graphs based on the computation of so-called *def-use* chains. A data dependence graph exposes the relations between statements of a program with respect to the *definition* and *use* (also sometimes called production and consumption) of data. However, when applying these principles to binary program analysis, this process is made difficult by the lack of accuracy of control flow graph techniques with respect to memory and register content, and the complexity of memory access patterns, which directly affects the accuracy of *def-use* chains. Each time such an access is incorrectly resolved, it breaks the chain into seemingly independent chains, and the data-dependence is lost between the corresponding statements. Therefore, the accuracy of the underlying memory model is *critical* in this context.

In order to cope with these limitations, we take inspiration from previous work [29] and leverage symbolic execution as a tainting engine.

4.3 Symbolic Execution

Dynamic symbolic execution is similar to dynamic emulation of a program with one main exception: instead of concrete ones and zeroes, the data in registers and memory are *symbolic variables*. This applies quite well to the evaluation of address leaks, since the variables in the expressions are retained through execution. For example, the x86 instruction `add eax, ebx` with `eax` containing the symbolic variable A and `ebx` containing the concrete value 1 will result in the expression $A+1$ being stored in `eax`. Later, we will use this detail of symbolic execution to identify leaked program pointers.

We utilize a standard symbolic execution engine, based on common techniques in the field of symbolic execution [11, 14, 31]. Our symbolic execution engine interprets binary code and applies actions carried out by this code onto a *symbolic state* containing the values in memory and registers. As the analysis comes to a conditional jump, the constraint solver is queried for possible values of the jump condition. When a condition can be both true and false (because the symbolic variable included in its expression is not constrained or loosely constrained), the state splits and the engine continues the analysis of both paths, with new constraints added to the variables involved in the comparison.

4.4 Symbolic tracking

Dynamic symbolic execution suffers from *path explosion*. As previously mentioned, when a jump conditional cannot be shown to be explicitly true or false, then both branches must be followed. As the analysis continues through a program, more and more of these branches are spun off (in fact, each conditional has an exponential effect on the number of states) until the analysis becomes unmanageable.

To remedy this, we use the dynamic symbolic execution engine to perform *selective symbolic tracking* by constraining our symbolic execution engine to analyzing only certain paths. Specifically, we only trace the previously-identified potentially leaking paths (as described in Section 4.1). Tracking these paths instead of exploring the entire binary keeps our analysis from experiencing a path explosion in unrelated code, and allows us to focus the analysis on detecting leaks. Conceptually, this is similar to the symbolic component of *concolic* execution, when concrete inputs are symbolically traced [18].

During symbolic tracking, we represent all pointers in the program (for example, the stack pointer, any pointers returned from allocation functions such as `malloc`, a code pointer into the binary program and libraries, *etc.*) as expressions representing the addition of a fixed offset to a symbolic variable representing the base of that region. This allows us to detect leaked pointers in the next step.

4.5 Symbolic execution as a tainting engine

The rich state information extracted from symbolic execution is used by *Sleak* to compute data dependence in a precise manner. More precisely, *Sleak* keeps tracks of potential variables of interest (*i.e.,* corresponding to addresses) by making these variables symbolic, and by symbolically executing the previously identified control-flow paths starting at each program point defining a variable of interest (which is determined through static analysis) and

until a sink (output function) is reached. In this context, we leverage under-constrained symbolic execution [28] in order to start execution at these arbitrary program points. We refer to this execution step as SE1. Similarly to [29], we only explore loops one time during this analysis step in order to speed up the execution while retaining taint information. In addition to this, SA1 also applies the inference rules defined in §3.4 in order to identify additional sources that were missed during static analysis.

When the execution reaches a sink, each function argument is analyzed. The way arguments are passed to output functions depends on the architecture, and the convention used by the compiler: arguments may be placed on the stack or placed in architecture specific registers. Figure 5 shows an example of function call on x86_64. For each argument passed to an output function, *Sleak* analyzes the corresponding symbolic expression, formulated as an Abstract Syntax Tree (AST), in order to determine eventual dependence on one of the previously marked symbolic variables corresponding to addresses.

```
mov        rax [rbp+8]
mov        rsi rax                ; arg 2
mov        rdi 0x40095d         ; arg 1
call        0x4005b0        ; printf@PLT
```

Figure 5: Calling an output function

When such a dependence exists, an address leak has been detected. For the sake of precision, the corresponding path is re-executed symbolically without the aforementioned optimizations tailored for tainting. The result of this analysis is the symbolic expression along with path constraints for each leaking path. We refer to this step of full symbolic execution as SE2.

From there, the process of reconstructing the original address based on the output is presented in Sections 4.6 and 4.7.

4.6 Recovering Address Information

Each path that is symbolically traced terminates at an output function. At this point, *Sleak* analyzes the final state of that path and checks the content of the output for the presence of symbolic variables corresponding to addresses. Let us describe it in an example. Consider the code in Figure 6.

```
// C code
int x = 1;
printf("X is \%d\n", &x);

// ASM translation
mov     [rsp+8], 1
lea     rsi, [rsp+8]
mov     rdi 0x40095d ; "X is \%d\n"
call    0x4005b0     ; printf@PLT
```

Figure 6: C and assembly code of type confusion stack leak.

In this example, the bug is introduced by the incorrect *referencing* of x: a pointer to it, rather than its value, is passed to `printf`. In the assembly code, this manifests as a `lea` (Load Effective Address) instruction, instead of the `mov` instruction that would be required to pass x by value. The `lea` instruction will, in this case, move `rsp+8` into the `rsi` register to pass it as an argument to `printf`. Since *Sleak* initializes the stack pointer to the symbolic value STACK_BASE at the beginning of the trace, `rsi` will contain some offset of that variable as well. For the sake of the example, let us suppose that the expression in `rsp` was STACK_BASE-32, in which case the expression in `rsi` will be STACK_BASE-24.

When *Sleak* detects that an expression that depends on one of the base addresses is passed to an output function, it analyzes it to identify what parts of the base address can be recovered. In this simple example, it is a trivial matter of subtracting 24 from `rsi` to recover STACK_BASE and defeat ASLR.

4.7 Addressing partial leaks

Not all arithmetic operations are directly reversible. While some operations such as adding/subtracting a constant or XORing a value with a constant are reversible, a number of other operations are not. *In these cases, bits of the initial value are lost*, which translates to multiple, possibly many solutions when trying to solve constraints on these expressions. *Sleak* leverages constraint solving to support a wide range of possible transformations on the pointer, including complex and/or irreversible arithmetic operations. By using constraint solving, our approach exercises the space of possible solutions, which constitutes the set of brute-force candidates for an attacker to use. Consider, for instance, the case of division on fixed-sized integers. This operation is not directly reversible. Let us assume that a program outputs an expression `out` as being $out = x/4$. When executing the program, if one observes that the output is 42, it is possible to obtain possible values for x using a constraint solver, *e.g.*, Z3, as follows:

```
z3.solve(out== 42, out== x/4)
```

Which will yield the solutions:`(x = 168, out = 42),(x = 169, out = 42),(x = 170, out = 42),(x = 171, out = 42)`

By leveraging constraint solving, *Sleak* is thereby able to automatically recover address information even in the case of subtle instances of partial pointer leaks (*e.g.*, arithmetic transformations over pointer values leading to leak only a few bytes), based on the knowledge of a concrete output value of the target running program.

4.8 Concolic tracing

Sleak supports concolic tracing, that is, the ability to feed the symbolic engine with concrete inputs obtained from the environment of a dynamic execution trace. Our concolic tracing module takes, as input, both symbolic data and concrete data obtained from the input state of the augmented CFG presented in Section 3.

At any point in the trace, the state of the program (including all of its memory and registers contents) can be obtained, and fed into our symbolic execution engine. This ability to switch from dynamic (concrete) execution to symbolic execution allows us to analyze code paths using a hybrid state composed of concrete and symbolic data. Symbolic data is introduced by the following three operations: 1) reading an address depending on a symbolic base, as presented in Figure 6, 2) reading unconstrained[9] data from the file system, the network, or any user input, 3) unsupported system calls.

[9]Since we cannot reason about such input values, these are represented as unconstrained symbolic variables at the time they are read from user input.

Sleak leverages this technique to analyze potential leaking paths identified during the first phase of the analysis.

5 EVALUATION

Our evaluation of *Sleak* comprises a set of userspace applications and services, a complex userspace library and a linux kernel filesystem. In more detail, we analyzed:

- 80 binaries from Capture The Flag (CTF) competitions including Defcon's final CTF and its qualifying events from years 2012 to 2018.
- `libXSLT`, a library specialized in the transformation of XML documents, which is part of many common software applications, including Firefox and Chrome. The extensive size and the complexity of this library made it a good candidate for evaluating our system.
- The overlayfs filesystem from the Linux kernel (used by Docker virtualization containers).

5.1 Ground truth data

We compared our results against (1) CTF writeups and (2) existing vulnerabilities published in the Common Vulnerabilities and Exposures database available from NIST, as well as from manual verification of source and binary code, which motivated our choice for open-source projects in our evaluation.

5.1.1 CTF binaries. We gathered 80 userspace services from prior capture the flag competitions. Since it is a requirement for successful exploitation in the presence of address space layout randomization, 4 of these services are vulnerable to information leakage, where a pointer address is leaked. We collected ground truth data from CTF writeups and manual reverse engineering.

5.1.2 libXSLT. Earlier versions of the `libXSLT` library were vulnerable to an information disclosure vulnerability, reported in CVE-2011-1202. This bug has been fixed since[10], after remaining unnoticed for 10 years[11]. Figure 7 shows the relevant basic block involving the unsafe operation which is causing the vulnerability (in the GenerateId function) at line 8. For additional clarity, the matching source code is presented in Appendix 8.2 in Figure 10. At the beginning of this basic block, the register `rbp` contains the value of a heap pointer `cur` (which corresponds to a pointer of type `xmlNodePtr`). This value is then copied into register `rdx` at line 2, multiplied by a constant at line 4, and shifted right by 6 at line 7. This is how the compiler translates and optimizes integer division (here, the value is divided by `sizeof(xmlNode)`).

5.1.3 OverlayFS. The implementation of the `overlayfs` filesystem in the Linux kernel (up to version 4.4.5) leaks the kernel memory address of a `struct dentry` pointer to userspace[12].

At the beginning of the vulnerable function, the register `rsi` contains a pointer to a `dentry` structure. This pointer is a heap pointer, that is allocated outside of the scope of this function. Therefore, by only looking at the code in Figure 8, there is no indication that

[10] `https://git.gnome.org/browse/libxslt/commit/?id=ecb6bcb8d1b7e44842edde3929f412d46b40c89f`

[11] It was introduced 10 years earlier in commit `1ad1ac261f5e4e0efbb656263a26d27be9ea2afe`

[12] This bug was fixed on September 16, 2016.

```
1  1   mov    rax, rbp        ; cur
2  2   mov    rdx, 8888888888888889h
3  3   lea    rsi,[rip+0x1257a] ; "id%ld"
4  4   mul    rdx
5  5   lea    rdi, [rsp+0x10] ; str
6  6   xor    eax, eax
7  7   shr    rdx, 0x6
8  8   call   0x7ffff7ba8920 <sprintf@plt>
```

Figure 7: Snippet (ASM) of the GenerateId function from libXSLT

```
1   1   push   rbp
2   2   push   rbx
3   3   mov    rcx,rsi
4   4   mov    rdx,0xffffffff81a85100 ;"format"
5   5   mov    rbp,rdi
6   6   mov    esi,0x14 ;"n"
7   7   sub    rsp,0x20
8   8   lea    rbx,[rsp+0x4]
9   9   mov    rax,QWORD PTR gs:0x28
10
11  10  mov    QWORD PTR [rsp+0x18],rax
12  11  xor    eax,eax
13  12  mov    rdi,rbx         ; "name"
14  13  call   0xffffffff813c5a60 // snprintf()
15  15  [...]
16  15  call   0xffffffff811966fe // printk()
```

Figure 8: Assembly translation of the source code in Figure 11.

this value is a pointer. This value is moved to register `rcx` (line 3) and the `snprintf` function copies it to the local stack variable at `rsp+0x4`. After some error checking (ommited from Figure 8 for brievity), the value of `name` is then exposed to userland by invoking `printk()`.

5.2 Experimental setup

Our system builds on top of the `angr` program analysis framework, which we extended (1500 lines of Python) with custom analyses and heuristics to detect information leakage vulnerabilities. Our dynamic trace collection mechanism is a custom implementation. We made light modifications to the Qemu emulator and leveraged its gdb stub to communicate with our analysis platform and to dump the memory of the stack, the heap, and the global data areas.

Our analysis runs on *stripped binary executables*. Therefore, in spite of the presence of open-source code in our evaluation dataset, *Sleak* operates identically regardless of whether the underlying code is open-source or closed-source.

For each binary, the number of basic blocks in the control-flow graph, the number of analyzed functions, and the number of sinks marked are logged. In the case of `libXSLT` and Overlayfs, we also leverage test cases to *collect dynamic traces*. For `libXSLT`, our dynamic phase consists of executing test cases which ship with the library. For the Linux kernel filesystem, it extracts a large archive containing the Linux kernel source tree while collecting dynamic traces. All our experiments were performed on a Dell Precision tower 5810 with 6 Xeon E5-1650 v4 @ 3.60GHz CPUs and 64GB of memory. See Section 6.1 for performance information.

Challenge	CFG nodes	Functions	Sinks	Leak	GT
CTF binaries					
0x00ctf_17_left	72	1	3	✓	✓
a5afefd29d5dc067ed6507d78853c691	496	16	11	✓	x
defcon_16_heapfun4u	200	5	1	✓	✓
ez_pz	91	2	3	✓	✓
pwn1	318	1	1	✓	x
int3rrupted	327	6	4	✓	✓
libXSLT	76842	505	27	✓	✓
Overlayfs	1981	191	27	✓	✓

Table 1: Analysis results (summarized)

5.3 Analysis results

We summarize our results (*i.e.,* we only report binaries for which a leak exists or is reported by our system) in Table 1, which we compare against the ground truth, represented in the last column labelled GT. *Sleak* successfully detects all instances of leaks present in our dataset, with the addition of two additional leaks which correspond to false positives, *i.e.,* where no leaks actually exist in the corresponding binaries. After investigation, we were able to determine that these false positives are caused by intentional stack manipulations attempting to obfuscate program behavior which do not follow standard practices (*i.e.,* as found in benign programs compiled with standard compilers). In all cases, *Sleak* returns the symbolic expression of the leaked variable along with its path constraints. *From this knowledge, an attacker who has observed a concrete output of the program can easily leverage constraint solving as described in §4.7 in order to reveal the solutions corresponding to the set of possible leaked address values (which corresponds to a single value in cases where a vulnerability exposes an entire address), and therefore bypass ASLR (i.e., a single address is sufficient to recover the base address of a loaded program or library).*

5.3.1 libXSLT. *Sleak* detects this vulnerability in *two different settings* involving two different *inference rules*: 1) in a *purely static* setting, without any knowledge of the dynamic behavior of the library, and 2) by starting the analysis from a *concrete state*, as described in §3.5. In the following, we present the results for both.

(1) Static detection. In a purely static setting, *Sleak* identifies 27 potential sinks during its first analysis phase (SA1), called from 7 distinct functions out of 505, as represented in Table 1, including the vulnerable call to `sprintf`. In terms of address detection (SA2), the `GenerateId` function does not present any information which would enable our analysis to determine that one of the parameters of `sprintf` is a pointer. The reason for this is that `generateId` gets this pointer *from its parameters*, which types are unknown to our analysis. However, during the SE2 analysis phase, while the symbolic execution engine explores multiple paths of the function, the analysis hits an IR operation corresponding to a dereference (*i.e.,* inference rule #3 from §3.4) in another function call happening in a parallel branch to the branch performing the vulnerable `sprintf` call. The call in question invokes the function `int xmlXPathCmpNodes()` which defeferences the same XMLNodePtr pointer. *Sleak* propagates this information back to the upmost branching point one function higher in the call tree, and to infer that the parameter of `sprintf` is indeed a pointer. This ability to propagate address type information *across branches* is powerful, and combined with address inference rules, allows us to reason about complex code.

(2) Static detection augmented with traces. When neither the analyzed code paths nor their parallel branches expose sensible IR operations, it may not be possible to infer address information from the code statically. Therefore, *Sleak* also leverages execution traces as presented in Section 3.5. By doing so, *Sleak* obtains a concrete call context for each function invoked in the execution trace, which is then used as part of our static model. The developers of `libXSLT` distribute the library along with test cases, which we leverage as input data in order to generate dynamic traces. Our system successfully detects the vulnerability, in the basic block represented in Figure 7. The value in register `rbp` is identified as a heap variable by our analysis, and the call to `sprintf` as a dangerous use of an address. *Sleak* infers the presence of an address due to inference rule #2 presented in §3.4.

Symbolic expression of leaked data Regardless of whether a static or dynamic detection method is used, when executing the detected path symbolically as part of SE4, at the beginning of the first instruction, our symbolic execution engine represents the content of register `rbp` as an unconstrained symbolic variable of the form `heap_addr_uid` where `uid` is a unique identifier. After executing instruction 3, the register `rsi` contains the format string `id%ld`. This represents the first argument to the call to `sprintf`[13]. When reaching the call to `sprintf`, *Sleak* parses the format string in register `rsi`, and determines that the value of the next argument can be fetched directly from register `rdx` without dereferencing a pointer, since `%ld` expects a long integer. At this point, the value contained in register `rdx` is represented by our symbolic engine by an expression describing operations on a symbolic heap pointer. This expression is represented internally in terms of bit vector operations by our symbolic engine, and it is equivalent to a division by `sizeof(xmlNode)` of the symbolic heap variable. Since arithmetic division is not a reversible operation, multiple solutions are possible (*i.e.,* it is a partial leak). As a result, *Sleak* outputs the *symbolic expression of the leaked address* along with its associated constraints.

5.3.2 Overlayfs. The Linux kernel presents a very asynchronous behavior as well as many dynamic characteristics, such as registering stuctures at runtime (*e.g.,*`struct inode_operations`) and other filesystem specific interfaces which are *hooked at runtime*. Therefore, the static detection phases of *Sleak* is not able to detect

[13] Following the calling convention of the ELF format on `x86_64`.

this vulnerability, but our approach of *static detection augmented with traces* is effective, following the same steps as the analysis of `libXSLT` described earlier. When reaching the call to `printk` at line 15 of Figure 8, our symbolic engine evaluates the content of register `rsi` to an unconstrained symbolic variable of the form `heap_addr_uid`, corresponding to the leakage of a full heap address. This expression indicates that the output value corresponds to an *entire address*.

6 DISCUSSION

6.1 Performance and scalability

In terms of scalability, a potential bottleneck of any approach relying on symbolic execution is peak memory usage. This is why Sleak carefully filters out the sets of paths to analyse based on static analysis. During our experiments, the peak memory consumption was below 20GB at all times, and the longest symbolic path was executed under 10 minutes. The overall analysis time was under 80 minutes for CTF binaries, and under 20 hours for libXSLT and the kernel filesystem together. We have not reached any memory bottleneck, even on kernel code, which demonstrates the effectiveness of our filtering approach.

6.2 Limitations

Sleak and its implementation rely on state of the art static analysis and symbolic execution technique. Despite this fact, our approach is subject to code coverage limitations and to state explosion, in some cases.

Code coverage: as discussed in Sections 3 and 4, the coverage of the static analysis techniques that we presented has some limitations. Due to the inherent difficulties of statically recovering a full control-flow graph, some parts of the analyzed binary application may not be reachable by *Sleak*, which potentially yields a number of false negatives. We mitigate this problem by leveraging dynamic resolution, as presented in Section 3. As a result, our approach provides a trade-off between coverage and accuracy.

State explosion: large code paths sometimes remain to be symbolically analyzed, even after the first phase of our analysis. This happens in particular when tracking heap pointers are initialized early, and used/leaked later in the program, after executing a long code path. Complex loops can also cause state explosion, regardless of the size of the analyzed paths. We partially mitigate this issue by leveraging veritesting and path prioritization techniques.

Implementation: empirically, while the implementation of our proof-of-concept is able to successfully analyze real-world binary applications, our modeling of the environment (*e.g.,* system calls) as well as some complex standard library functions (*e.g.,* string functions such as `strcmp`) is neither comprehensive nor completely accurate, which may impacts our coverage in some cases (*e.g.,* by involving incorrect constraints on some symbolic variables, which may lead to unsatisfiable symbolic states.) This problem is not specific to our approach, and affects any binary analysis approach in general.

7 RELATED WORK

Information leakage. In the last decade, researchers have proposed several approaches to detect information leaks from kernel-space towards user-space at the source code level. Peiro *et al.* recently proposed an approach based on static analysis for detecting kernel stack-based information leaks [27]. The proposed approach is able to detect particular instances of information leaks (such as leaks caused by missed memory initializations and missing checks on user reads). The authors found five new vulnerabilities in the Linux kernel by analyzing its source code with their tool. However, one of the limitations of this approach is that the analysis is limited to single functions (*i.e.,* inter-function analysis is not supported).

Johnson *et al.* [22] previously introduced a pointer bug detection model based on type qualifier inference. The authors extended the open-source tool CQUAL [2] for this purpose. This approach requires manual annotations of functions, such as system calls accepting user arguments, before the analysis can be performed. While such source-level approaches address a similar problem than the focus of our study, these are not applicable as-is at the binary level, due mainly to the lack of type information in the disassembly.

On the information theory side, past research focused on quantitative information flow [6, 8, 38], as initially proposed by Denning [30], where the objective is to measures the amount of secret information leaked by a program, by observing its outputs. In particular, Backes *et al.* [8] proposed a model based on equivalence relations to characterize partial information flow. This model is used to identify which information is leaked and provide a quantitative interpretation of the leaked variables. While this approach is generic and may be used to reason about information leakage of any sort, it does not focus on individual program runs, but instead generalizes the behavior of a program. In comparison to our approach, this model lack information about the mapping between actually leaked bits, and their origin within the program's memory.

In [32], Seibert *et al.* present a model for remote side channel attacks that allow attackers to exploit memory corruption vulnerabilities even in the presence of code diversification. The authors claim that the assumptions behind code diversity are broken, due to the fact that executing code leaks information about the code itself, which may allow the attacker, under certain circumstances, to recover the location of code gadgets. While this model achieves goals similar to ours, this requires the attacker to actively modify the state of the program, by overwriting data or crafting specific inputs, where our approach is *passive* and relies solely on the observation of the outputs of the program.

Type casting verification. Existing approaches to detect information leakage either focus on out-of-bound reads, type confusion or use-after-free using dynamic approaches or require source code [16, 21, 24, 26, 33]. In particular, Hextype leverages source-code analysis along with compiler-level techniques in order to replace static checks by runtime checks. Similarly, EffectiveSan [16] enforces type and memory safety in C and C++ programs by using a combination of low-fat pointers, type meta data and type/bounds check instrumentation.

However, as of today, little attention has been spent in the research community to address the detection of *indirect* information leaks in binary software.

Binary program analysis. At the binary level, a number of approaches based on symbolic execution have been proposed to detect memory corruption vulnerabilities in off-the-shelf applications [31, 39, 41], to analyze firmware [37, 44] in order to detect backdoors and logic bugs, and to analyze drivers [13, 23] for reverse engineering purposes and for detecting undesired behavior. Among these existing binary-level approaches, the BORG [26] focused on detecting buffer overreads in binary software using guided symbolic execution. Our approach draws on similar concepts, but focuses on addressing different challenges.

8 CONCLUSION

We presented *Sleak*, a system designed to recover information about the memory layout applications, even in the presence of address space randomization (ASLR). Our system analyzes applications at the binary level, and detects information disclosure vulnerabilities, regadless of how many bits of pointer addresses are leaked. *Sleak* leverages symbolic execution to craft precise symbolic expressions representing the addresses of known objects of the target application, such as stack or heap variables, or function pointers. As a result, even in the case of partial information disclosure, *Sleak* is able to recover useful information about the leaked address, and defeat ASLR.

AKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This material is based on research sponsored by DARPA under agreement numbers HR001118C0060, FA8750-19-C-0003 and the NSF under Award number CNS-1704253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government.

REFERENCES

[1] Angr, a binary analysis framework. http://angr.io.
[2] CQUAL, A tool for adding type qualifiers to C. http://www.cs.umd.edu/ jfoster/cqual/.
[3] OpenBSD's W^X. http://www.openbsd.org/papers/bsdcan04/mgp00005.txt.
[4] The PAX Team. https://pax.grsecurity.net.
[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[6] M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, and C. Palamidessi. Foundations of security analysis and design vi. chapter Quantitative Information Flow and Applications to Differential Privacy, pages 211–230. Springer-Verlag, Berlin, Heidelberg, 2011.
[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.
[8] M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.
[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.
[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242, May 2014.
[11] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.
[12] M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. *Security Privacy, IEEE*, 12(4):63–67, July 2014.
[13] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 167–180, New York, NY, USA, 2010. ACM.
[14] V. Chipounov, V. Kuznetsov, and G. Candea. *S2E: A platform for in-vivo multi-path analysis of software systems*, volume 47. ACM, 2012.
[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.
[16] G. J. Duck and R. H. Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195. ACM, 2018.
[17] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.
[18] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.
[19] W. Herlands, T. Hobson, and P. J. Donovan. Effective entropy: Security-centric metric for memory randomization techniques. In *Proceedings of the 7th USENIX Conference on Cyber Security Experimentation and Test*, CSET'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association.
[20] E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 1–8, New York, NY, USA, 2011. ACM.
[21] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387. ACM, 2017.
[22] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.
[23] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.
[24] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, pages 81–96, 2015.
[25] H. Marco-Gisbert and smael Ripoll-Ripoll. Exploiting linux and pax aslr's weaknesses on 32- and 64-bit systems. In *Blakhat Asia 2016*, 2016.
[26] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 87–97, New York, NY, USA, 2015. ACM.
[27] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. Detecting stack based kernel information leaks. In *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, pages 321–331. Springer, 2014.
[28] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, pages 49–64, 2015.
[29] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Bootstomp: on the security of bootloaders in mobile devices. In *26th USENIX Security Symposium*, 2017.
[30] D. E. Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.
[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.
[32] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 54–65, New York, NY, USA, 2014. ACM.
[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.
[34] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.
[35] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307,

New York, NY, USA, 2004. ACM.
[37] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.
[38] G. Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.
[39] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.
[40] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.
[41] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the network and Distributed System Security Symposium*, Feb. 2011.
[42] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 17–31, Berlin, Heidelberg, 2006. Springer-Verlag.
[43] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.
[44] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, February 2014.

APPENDIX

8.1 Function identification

Binaries not exposing symbol information (*i.e,* stripped binaries), do not provide information about the location of functions, which prevents *Sleak* from directly identify output functions. However, the general problem of function identification in binary remains an open research challenge, and the accuracy of existing solutions is limited [9, 20]. Our approach to function identification takes different steps, and focuses on the particular case of detecting output functions. We first identify function boundaries by scanning the binary for function prologues and epilogues. Once the location of functions has been identified, we use the following heuristics to recognize output functions: if one of the identified function is invoking the `write` system call, and if one of the function's arguments is passed to this system call, we consider it an output function. This allows us to detect the majority of output functions, given the fact that most of them are, in practice, implemented as writing to an underlying file descriptor [14], and therefore perform a call to `write`.

8.2 Network authentication example

The source code of the network authentication function presented in §2 is shown in Figure 9 below. The vulnerability lies at the end of the function `verify_token`: when the authentication is unsuccessful, an unprotected access to the union causes the value of its `key` member to always be read regardless of the token's type. If a token with `type_pass` is passed to the verification function, this results in an address leak, where a `char *` address is interpreted as being an `int` value. The amount of leaked bits depends on the architecture[15]. Looking at the C code in Figure 9, it is possible for an analyst to manually determine that the unguarded call to `sprintf` may leak an address. However, analyzing code in binary form, as shown in Figure 3, is much less intuitive than the source code version of the same program, due to the lack of type information about the data structures.

```
1  int authenticate(token *v)
2  {
3      ...
4      switch(v->type)
5      {
6          case type_pass:
7              // Handle password checking.
8              break;
9          case type_key:
10             // Handle key verification.
11             break;
12         default:
13             return -EINVAL;
14         }
15 }
16 int verify_token(token *v)
17 {
18     int auth;
19     ...
20     if (!v)
21         return;
22     ...
23     auth = authenticate(v);
24     if (auth == 1)
25     {
26         v->status = valid;
27         return;
28     }
29     ...
30     sprintf(buf,"INVALID:%d",v->auth.key);
31 }
32 int authenticate_client(void)
33 {
34     token *tk;
35     ...
36     tk = received_token()
37     verify_token(tk);
38 }
```

Figure 9: Accidental address leak through type confusion.

9 GROUND TRUTH/SOURCE CODE

9.1 libXSLT

Let us consider the code in Figure 10. This code snippet shows the vulnerable version of the function providing the XSLT transformation `generate-id()`. For the sake of brevity, we only represented the relevant parts of the function, *w.r.t.* the aforementioned vulnerability in Figure 10. The purpose of this function is to generate a unique identifier for a given XML node. In this function, the variable `val` represents the unique identifier. In order to generate a unique value, the value of `cur` is assigned to `val`, which is then divided by `sizeof(int)`. Note that while `cur` is a stack variable, its content is the address of `ctx->context->node`, which contains the address of a heap pointer. The identifier contained in `val` is then appended to a string, which value is later exposed in the output document. Exploiting this bug reveals the heap location of the process running

[14] This model does not consider hardware-specific I/O mappings implemented through *e.g.*,`ioctl`.

[15] On `x86_64` machines, this corresponds to a leak of 32 bits out of a 64-bit address.

this library. Among other applications using `libXSLT`, Chrome, Safari and Firefox were affected. This vulnerability was introduced by an intentional, but unsafe, pointer manipulation. Without security in mind, an address may seem like a good candidate, for its uniqueness, to generate an identifier. Looking at the comments in the vulnerable version of the code, we can also confirm that the programmer intentionally used the address of the XML node to compute a unique identifier. However, by multiplying the output value by `sizeof(int)`, it is possible to fully recover the leaked heap address.

```
   void GenerateId(ContextPtr ctxt, int nargs)
   {
1      xmlNodePtr cur = NULL;
2      unsigned long val;
3      xmlChar str[20];
4
5      cur = ctxt->context->node;
6      val = (unsigned long)(char *)cur;
7      val /= sizeof(xmlNode);
8      sprintf((char *)str, "id%ld", val);
    }
```

Figure 10: Simplified version of CVE-2011-1202 in libXSLT.

9.2 OverlayFS

In recent years, because of the escalation of user-space security mechanisms and protections against memory corruption attacks, it became more and more difficult for attackers to reliably execute their exploits on remote systems. In contrast, the Linux kernel started to implement support for address space layout randomization later, and even when it is enabled, it provides less entropy than its user-space counterpart. As a result, attackers' attention partially shifted towards kernel-exploits, which became more common.

```
struct dentry *ovl_lookup_temp(
                struct dentry *workdir,
                struct dentry *dentry)
{
1 struct dentry *temp;
2 char name[20];
3
4 snprintf(name, n, "#%lx", (unsigned long)
                                    dentry);
5 temp = lookup_one_len(
                name, workdir, strlen(name));

7  if (!IS_ERR(temp) && temp->d_inode)
8  {
9    pr_err(msg, name);
10   dput(temp);
11   temp = ERR_PTR(-EIO);
   }
        return temp;
}
```

Figure 11: Vulnerable function in the overlayfs filesystem.

9.2.1 Pointer disclosure. The implementation of the `overlayfs` filesystem in the Linux kernel (up to version 4.4.5) leaks the kernel memory address of a `struct dentry` pointer to userspace[16], as shown in Figure 11. This pointer address was meant to be used as a unique identifier (and to that extend, this bug is similar to the bug in `libXSLT` that we presented earlier).

[16]This bug was fixed on September 16, 2016.