DESENSITIZATION: Privacy-Aware and Attack-Preserving Crash Report

Ren Ding*, Hong Hu*, Wen Xu, Taesoo Kim
Georgia Institute of Technology
{rding, hhu86, wen.xu, taesoo}@gatech.edu

***Abstract*—Software vendors collect crash reports from end-users to assist in the debugging and testing of their products. However, crash reports may contain users' private information, like names and passwords, rendering the user hesitant to share the reports with developers. We need a mechanism to protect users' privacy in crash reports on the client side while keeping sufficient information to support server-side debugging and analysis.**

In this paper, we propose the DESENSITIZATION technique, which generates privacy-aware and attack-preserving crash reports from crashed executions. Our tool adopts lightweight methods to identify bug-related and attack-related data from the memory, and removes other data to protect users' privacy. Since a large portion of the desensitized memory contains null bytes, we store crash reports in spare files to save the network bandwidth and the server-side storage. We prototype DESENSITIZATION and apply it to a large number of crashes of real-world programs, like browsers and the JavaScript engine. The result shows that our DESENSITIZATION technique can eliminate 80.9% of non-zero bytes from coredumps, and 49.0% from minidumps. The desensitized crash report can be 50.5% smaller than the original one, which significantly saves resources for report submission and storage. Our DESENSITIZATION technique is a push-button solution for the privacy-aware crash report.

I. INTRODUCTION

Software vendors collect crash reports from their end-users to improve the stability and security of their products [2], [5], [36], [58], [79]. The crash report is either a coredump file that captures the CPU context and the memory content of the crashed program [1], [51], or the program input that makes the execution crash [19], [80], [24], [4]. Unfortunately, in either case, the crash report may contain privacy-sensitive information of individual users. For example, a recent study on 2.5 million crash reports of a popular web browser identifies an enormous amount of private user data, including user names, passwords, session tokens, and personal contact information [69]. A crash report with private information is unwelcome to both users and developers. While users are unwilling to share the crash reports due to the concern of leaking their private information, developers do not want to collect privacy-sensitive reports where any storage breach may lead to bad publicity, and even financial liability.

To protect users' privacy while supporting timely bug fixing, we need to remove users' sensitive data from crash reports before sending them to developers. We call this process DESENSITIZATION. Several proposals aim to *desensitize* the crash report, but unfortunately, we have not seen any real-world deployment. Techniques like Scrash [13] rely on developers to manually annotate sensitive data to remove them from crash reports. However, the manual annotation is usually time-consuming and error-prone, and does not work on closed-source programs. Pattern-based search is a general method to identify particular privacy-sensitive data, like URLs and email accounts [69], but it cannot handle program-specific data. Another set of works aims to desensitize the crashing input, which should make the program crash in a similar manner as the original [19], [80], [24], [4]. However, these proposals rely on computation-heavy techniques, such as symbolic execution [15], [22], [20], [16] or taint analysis [76], [61], [68], causing significant burdens to users or developers. More importantly, they require a non-trivial change to the current crash-report systems, which mainly use crash files instead of crashing inputs to classify and prioritize different bugs [2], [5], [36], [58], [79].

Given the understanding of the current obstacles, we identify three desirable properties that a DESENSITIZATION technique should have for wider deployment:

- **Privacy-aware**: The technique removes users' private information from the crash report as much as possible.
- **Attack-preserving**. The desensitized crash report still contains sufficient information for effective bug analysis. If the execution crashes due to a failed attack, the technique should keep the attack residue in the report.
- **Widely applicable**. The technique should be lightweight for end-users and compatible with current crash-report systems. It should work on closed-source programs, cover various types of sensitive data, and generate minimal crash reports for resource-limited environments.

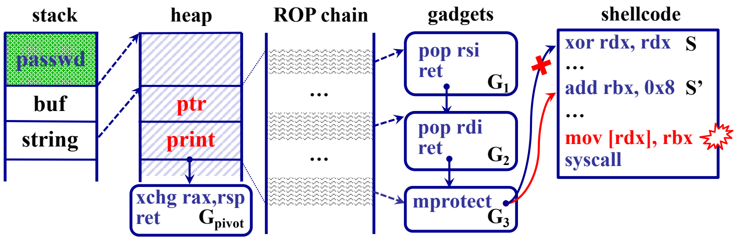
In this paper, we propose a bug-oriented and attack-oriented method to desensitize crash reports so that they capture a snapshot of the crashed program with minimal risk of leaking users' private information. Our observation is that although there are many bugs on the client side, only a few general techniques are used on the server side to diagnose bugs and detect attacks. For example, call-stack-based analysis is widely used to classify and prioritize crash reports [57], [8]. Therefore, DESENSITIZATION keeps only the information for commonly used bug-analysis techniques and removes all other data to protect users' privacy. We design one general and four lightweight techniques to identify bug-related and attack-related data from the crashed memory snapshot. The general technique

*The two lead authors contributed equally to this work.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24428
www.ndss-symposium.org

```c
1  #define MAX_LEN 64
2  typedef struct { char *ptr; void (*print)(); } String;
3  void printString() { ... }
4  void vuln(char *input) {
5    char passwd[MAX_LEN];   load_passwd(passwd);
6    char *buf = (char   *) malloc(MAX_LEN);
7    String *string = (String *) malloc(sizeof(String));
8    string->ptr = buf; string->print = &printString;
9    strcpy(buf, input);
10   string->print();
11 }
```

(a) Vulnerable source code.



(b) Memory layout before crash.

Fig. 1: A vulnerable code snippet and the memory layout before the execution crashes. The code has a heap-based buffer overflow bug at line 9. Attackers corrupt a function pointer to pivot the stack, launch ROP attacks, and eventually run the shellcode. However, due to the incorrect address, the program crashes and generates a coredump. The stack contains user password — a sensitive private data value.

scans the memory snapshot to identify all pointers, as most crashes and attacks stem from corrupted pointers. For each type of prevalent bug and attack, we design one technique to identify specific information or tailor existing techniques for client-side efficiency. Specifically, we design a heap module to identify all heap metadata, a format string module to detect malformed format strings, an ROP module to search for suspicious ROP gadgets, and a shellcode module to collect possible payload. The modular design makes our tool extensible to support future bug-analysis and attack-detection techniques.

The prototype of our Desensitization technique can generate desensitized crash reports from either crashed executions or existing reports. For the runtime report generation, our tool installs a signal handler to take over the execution once the process crashes. It applies Desensitization on the memory snapshot and generates the desensitized crash reports that are compatible with existing formats, such as coredumps (the default format on Linux [51]) and minidumps (the default format on Windows and many large programs, like Firefox [1]). It can also desensitize existing reports to remove private information. Currently, our prototype supports both 32-bit and 64-bit Linux ELF programs. We plan to add the support of Windows PE binaries in future work.

To understand the efficacy and efficiency of our tool on protecting users' privacy, we evaluate it on four real-world applications and one CTF (capture-the-flag) challenge: the multimedia player `ffmpeg`, the PHP language interpreter `php`, the JavaScript engine `chakra`, the web browser `firefox`, and the DEFCON 2015 challenge `tachikoma`. We collect 13,390 crash reports from 86 known bugs and emulated attacks, including 11,875 coredumps and 1,515 minidumps. The result shows that on average, Desensitization can remove 80.9% of all non-zero bytes from each collected coredump and remove 49.0% of that from each minidump. Further, Desensitization helps reduce the file sizes of coredumps by half, which saves network bandwidth for end-users to report the bug and reduces disk usage for developers to store the reports. Meanwhile, the desensitized crash reports still contain sufficient information to support common bug-analysis and attack-detection techniques. Specifically, the call-stack-based analysis produces the same result for reports before and after applying Desensitization. For example, we submit the original and desensitized reports to `socorro` – the crash-analysis framework of `firefox`. The platform groups them into the same group, indicating that our tool keeps the necessary information for bug classification. Our manual inspection on several attack-caused crashes confirms that Desensitization keeps all attack residues necessary for developers to analyze these attacks. Desensitization can complete its work with reasonable resources, taking less than 15 seconds process each crash. Therefore, Desensitization is a practical yet effective method to generate privacy-aware and attack-preserving crash reports.

Type	Semantics	Examples
Cookies	website tokens	`Geo=(4.3267961,52.096922)`
Forms	current user-input	`<form ... enctype="multipart-formdata">`

Autofilling	user-input history	`<form ... autocomplete="on">`
		`Passwd: <input ... autocomplete="on">`

TABLE I: Possible privacy leakage from minidumps of `firefox`, including website tokens and user inputs. See §V-B3 for more details.

We make the following contributions in this paper:

- **Privacy-first crash report**. We design a new format of crash report that can protect users' privacy while retaining sufficient information for server-side bug detection and attack analysis. The new format is compatible with most mainstream crash-report systems.
- **Techniques to generate a new report**. We implement one general and four specific techniques to construct crash reports in the new format from crashed processes. Our tool is easily extensible for future analysis techniques.
- **End-to-end system**. Our evaluation shows that our tool is practical for generating crash reports that protect user privacy while keeping attack residues.

Desensitization is a push-button solution for privacy-aware crash reports; developers just have to include the library of Desensitization in their products. We have released the source code of our tool and the collected crash reports at https://github.com/sslab-gatech/desensitization.

II. Problem Definition

In this section, we first present one example to illustrate the Desensitization problem. Then we define our threat model.

A. *Motivating Example*

We inspect the source code of `firefox` and its runtime, and identify several scenarios in which users' private information

Techniques	CallStack	IP	Signature	RevExec
Adobe-CR [2]	✓		✓	
Apport [78]	✓		✓	
Backtrace [8]	✓		✓	
Chromium-CR [36]	✓		✓	
CREDAL [81]				✓
Crash Graphs [45]	✓			
KLEE* [15]	✓	✓		
Liblit *et al.* [48]	✓	✓		
Mac-CR [5]	✓		✓	
Modani *et al.* [56]	✓			
POMP [82]				✓
Rebucket [27]	✓			
REPT [25]				✓
RETracer [26]				✓
Schroter *et al.* [70]	✓			
Socorro [57]	✓		✓	
WER [33]	✓		✓	
!analyze* [55]	✓		✓	
Desensitization	yes	yes	yes	

TABLE II: Crash analysis and triage techniques, and underlying mechanisms. **CallStack** relies on call stacks at the crashing point to find unique bugs; **IP** uses the crashed instruction address to deduplicate crashes; **Signature** adopts program-specific heuristics to the callstack techniques for more effective triage. $*$ indicates the uncertainty due to the lack of proper documentation found. The last line shows whether or not the desensitized crash reports still support these techniques.

Schemes	Heap	Struct	ROP	Shcode
CaVer [47]		✓		
DangNull [46]	✓			
HOTracer [42]	✓			
KOP [17]		✓		
Polychronakis *et al.* [66]				✓
Polychronakis *et al.* [65]				✓
ROPecker [21]			✓	
ROPMEMU [37]			✓	
ROPscan [67]			✓	
SBCFI [63]			✓	
ShellOS [75]			✓	
SigGraph [49]		✓		
Desensitization	yes	yes	yes	yes

TABLE III: Anomaly detection schemes and their detection targets. The last line shows whether or not the desensitized reports still contain necessary information to identify these attacks.

is dumped into the crash report. Table I provides a summary of our findings, where website cookies, user-inputs, and input history could be leaked. We leave the detailed discussion of these cases in §V-B3. Instead, we use a simplified example to demonstrate the privacy issue in crash reports.

Figure 1 shows a vulnerable program that is exploited by attackers, but the attack fails due to an invalid memory access. The code in Figure 1a first loads the user's password to a stack buffer `passwd` (line 5). Then, it allocates two heap objects, `buf` and `string`, and initializes their fields accordingly (line 6-8). However, the `strcpy` at line 9 may overwrite the heap memory beyond `buf` if the untrusted `input` has more than `MAX_LEN` bytes, leading to the corruption of the `string` object. In that case, the indirect function call at line 10 may divert the control-flow to an attacker-expected location. Figure 1b shows the memory layout of a failed attack, where attackers overwrite the function pointer `string->print` with the address of a stack-pivot gadget. The gadget changes `rsp` to the attacker-controlled memory region, which contains a sequence of code addresses of ROP gadgets. After running several gadgets, the attack finally executes the shellcode. Due to the incorrect address, the execution crashes at an invalid memory access. The operating system or a user-space library (*e.g.*, breakpad [35]) will generate a crash report that captures the memory status. Once the user agrees, the crash report will be sent to developers for bug analysis.

Privacy Concern. The crash report may contain users' private information. For example, the stack variable `passwd` in Figure 1b contains users' password, which should be confidential. However, current crash-report systems include all memory content in the coredump or at least all stack content in the minidump, which will leak the users' password to developers. In a recent study, researchers identified 20,000 session tokens and 600 passwords from browser crash reports, demonstrating the severity of the privacy leakage issue [69]. The straightforward solution is to remove all privacy-sensitive data from the crash reports before sending them to developers. However, existing works either rely on developers to manually annotate such variables or depend on non-scalable inter-procedural data-flow analysis to find them [13]. We need an effective and efficient solution to eliminate sensitive data from crash reports.

Attack/Bug Residue. To support bug analysis and attack detection on the server side, the crash report should contain sufficient information of the bug or residue of the attack. For example, in Figure 1b, we can find various data pieces used by the attacker, including the address of the stack-pivot gadget G_{pivot}, several ROP gadget addresses (*e.g.*, G_1, G_2, G_3) and the malicious shellcode S. With this information, developers may determine the reason for the crash — a failed exploit against a heap-based buffer overflow. They can also identify the exploitation methods used in the attack, like stack pivoting, ROP, and shellcode injection. Such information helps developers pinpoint the bug location and enables them to correctly estimate the bug severity (*i.e.*, highly risky). Finally, they can fix the bug by patching line 9 to avoid future crashes and attacks.

As we can see from the example, a Desensitization technique contains two contradictory goals: one is to reduce the user information from the crash report as much as possible to protect the user's privacy; another goal is to keep as much information as possible to afford effective bug diagnosis and attack analysis. Our work starts from widely used bug-analysis techniques, and aims to generate a minimal crash report that supports developer-side bug analysis and repair.

B. Bug-analysis and Attack-detection Techniques

We performed a study of existing bug-analysis and attack-detection tools used on the server side, aiming to understand the commonality of these techniques. Table II and Table III show our study results about crash-classification techniques and exploit-detection tools. Generally, most classification techniques rely on either call stacks, program counters, or customized signatures at crash sites for efficient triage. For example, `socorro` [57], developed and used by the `firefox` team, combines call-stack information and several program-specific heuristics to group different crash reports. In addition, heavier techniques, such as reverse execution [25], [82], [26], [81], are also proposed to examine crashes in greater detail. Among all anomaly-detection mechanisms in Table III, four types of information are commonly used to detect attacks, *i.e.*, the heap metadata, program-specific structures, ROP gadgets, and shellcode. For example, several works aim to detect ROP attacks by identifying ROP gadgets during program execution or from the execution trace [21], [37], [67], [63], [75].

ROPscan as an example. ROPscan [67] takes two steps to identify ROP attacks from network packets or memory buffers: it first sequentially scans every 4-byte value from the input to find potential code pointers that point to executable code locations (assuming a 32-bit system); it then starts to execute the pointed-to code and identifies gadgets based on the length of each block and the target of the ending instruction. If the number of distinct gadgets exceeds the threshold, ROPscan concludes the input contains a ROP payload. ROPscan can detect ROP gadgets from the coredump generated in Figure 1b. Specifically, given the whole memory snapshot captured in the coredump, ROPscan first identifies the 4-byte value G_{pivot} pointing to the executable code section. However, it cannot concretely execute this gadget, as the value of `rax` is missing. Then, it continues to identify the next code pointer G_1. After that, it tries to execute the code at G_1, which will reveal more gadget addresses, *i.e.*, G_2, G_3. ROPscan treats these three consecutive ROP gadgets as an indicator of the ROP attack.

We design Desensitization to retain adequate information in the crash report to support as many server-side analysis techniques as possible, while keeping the client-side computation lightweight to avoid heavy burden on end-users. Although existing techniques may have different algorithms to parse data, some types of data are widely used by most techniques. Based on this observation, Desensitization chooses to focus on collecting common data to support popular analysis techniques with minimal overhead.

C. Threat Model

We assume a software developer who creates benign but potentially buggy programs, and an end-user who likes to use the program but cares about her privacy – she does not want to share any credentials with the developer. The program execution may trigger some program bugs, either accidentally by the user's rare operation or intentionally by an attacker. Due to the bug, the execution terminates on the user side. The crash report system embedded in the buggy program, or provided by the underlying operating system, generates a crash report that describes the execution failure. The user likes to share the crash report with the developer to help diagnose the bug and fix it in time. However, she has the concern that her private information in the crash report may be leaked to the developer or to others through the developer's activity. The developer does not want the liability of protecting the user's privacy but is eager to use the crash report to debug the program.

III. Design

Our Desensitization technique identifies necessary information that is commonly related to bugs and attacks. First, it scans the whole memory to identify pointers (§III-A), including code pointers and data pointers. Our observation is that most crashes are caused by corrupted pointers, while many attacks manipulate pointers to achieve final goals. Second, our technique tackles other data that can provide debugging aids to track down multiple types of vulnerabilities (§III-B). For example, heap metadata is useful to identify heap overflows and use-after-free bugs, and therefore we follow the design of each heap manager to find all heap structures. Third, Desensitization considers possible exploit residues in the memory and uses heuristic-based methods to identify the related data of popular attack vectors. Table IV provides a list of information collected by Desensitization and potential vulnerabilities and attack vectors that are related to the collected information. Note that some vulnerabilities or attacks require a combination of several modules of Desensitization to get sufficient information. For example, the detection of dangling pointers requires both data pointers and heap metadata. Other than embedded techniques, our tool also provides convenient APIs for developers to write customized Desensitization modules based on their domain knowledge, like removing known useless pointers for analysis to avoid potential information leakage or keeping more data that are critical to detect bugs (§III-C).

Module	Collected Data	Related Bugs & Attacks
Pointer	`Code ptrs`	ROP [73], JOP [11], COP [18], GOT.PLT corruption, vtable injection [71], etc.
	`Data ptrs`	DOP [40], type confusion [47], use-after-free [46], double-free [46], out-of-bound, etc.
Heap	`Chunk size`	heap overflow, overlapping chunks [74], heap spray, etc.
	`PREV_IN_USE`	use-after-free [46], double-free [46], unsafe-unlink, bin-dup, house-of-* series [74], etc.
ROP	`Gadgets & args`	ROP [73], JOP [11], COP [18], etc.
Fmtstr	`Strings & args`	format string attack
Shcode	`Payloads`	shellcode injection

TABLE IV: Information collected by Desensitization, and potentially covered vulnerabilities and attacks. Our system supports easy extension for developers to customize Desensitization for a program-specific crash format, like nullifying more pointers to save space or keeping more data for debugging purposes.

System Architecture. Figure 2 shows the architecture of our Desensitization technique and its position in current crash-report systems. Desensitization is deployed on the client side, either as an exception handler of the program, or adopted by the operating system as a system-wide crash-report generator. Once a process crashes, Desensitization is invoked with the full memory space as the input. In current systems, all memory content is directly dumped in the crash reports. Instead, Desensitization takes several lightweight analyses to identify bug-related and attack-related data from the memory (shown as a small rectangle under each technique). Then, our technique keeps all identified data in the memory and nullifies others to remove the user's private information. Optionally, we store the desensitized memory as a sparse file and compress it to reduce its size (not shown in the figure). Finally, our technique utilizes existing systems to deliver the crash report to the server through the network. On the server side, developers can use any heavy technique to analyze the crash, like taint analysis [76], [61], [68], symbolic execution [15], [22], [20], [16] or even manual inspection. Finally, they determine the reason for the crash and synthesize a patch to fix the bug.

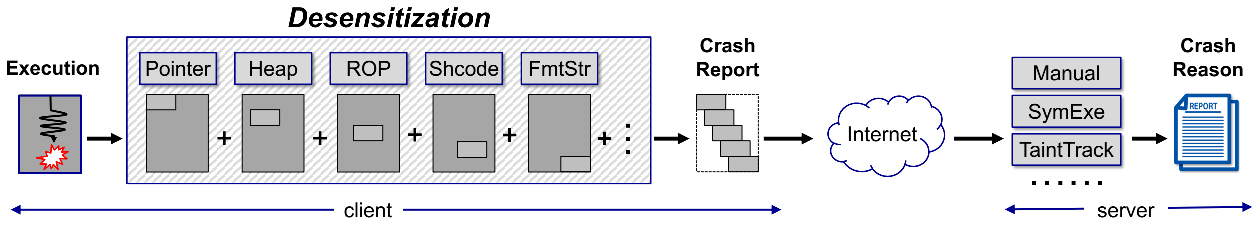


Fig. 2: Overview of DESENSITIZATION and its position in current crash-report systems. Once a process is crashed on the client side, our lightweight tool will inspect the process memory to identify bug- and attack-related data, like pointers. It removes all other data from the memory to generate a privacy-aware crash report. The report is sent to servers through network and is stored on the server side. Developers will use heavy bug-diagnosis and attack-analysis techniques to find the root cause of the crash.

A. *Pointer Identification*

Pointers play important roles in the program's normal execution, bug diagnosis, and attack detection. Code pointers hold the addresses of code segments, which enable efficient implementations of dynamic behaviors, like callbacks and polymorphism of object-oriented programming. Data pointers connect program variables together and support efficient memory accesses. Therefore, many debugging efforts start by examining pointers in a crash report. For example, call-stack-based analysis recovers the call stack by following stack pointers and base pointers [57], [8]. Meanwhile, pointers are common targets of attack techniques, including both traditional attacks such as ROP and format string attacks, and recently prevalent use-after-free attacks and type confusions.

DESENSITIZATION adopts generic approaches to identify all pointers from the memory. Specifically, it examines every pointer-size bytes in the crashed memory, *e.g.*, 4 bytes for x86 systems and 8 bytes for x86-64 systems, and checks whether they are potential code or data pointers. DESENSITIZATION considers a pointer-size memory as a pointer *if and only if* its value falls into one valid region of the memory space with proper access permissions. We treat a pointer as a code pointer if the pointed page has the execution permission; otherwise, we label it as a data pointer. Values pointing to any invalid memory range, or non-accessible memory region, like those reserved for dynamic allocations, will be considered as non-pointers. This method will not miss any pointers in memory, but may include non-pointers in the crash report. Fortunately, our evaluation shows that even with potential false positives, it still significantly reduces the size of the crash report.

We adopt several optimizations to improve the performance of memory scanning. First, DESENSITIZATION combines consecutive memory regions regardless of their permissions (as long as accessible) to reduce the number of comparisons when validating pointers. For example, crashes of `firefox` contain 340 memory regions, leading to heavy comparisons to validate each pointer-size value. Merging continuous memory reduces more than half of these regions, significantly improving the validation performance. Second, our scanner maintains the merged memory regions as an interval tree, which searches address ranges in a logarithmic time complexity.

Capturing GOT.PLT. ELF binaries use the Global Offsets Table (*GOT*) to access the functions and variables of external libraries. Specially, the addresses of external functions are stored in the GOT.PLT section (PLT for Procedure Linkage Table), one entry for each function. A GOT.PLT entry originally refers to an instruction of the PLT section and is changed to the real address of the external function after its first invocation. These entries are common corruption targets for attackers to bypass randomization-based defenses [30]. Our pointer identification will collect all normal GOT.PLT entries as valid code pointers. Even if one GOT.PLT entry gets corrupted and becomes an invalid pointer, DESENSITIZATION still strives to preserve it due to its sensitivity toward bug-related and attack-related debugging. Specifically, if DESENSITIZATION can identify the GOT.PLT section from the dump information, it will simply keep all entries in the section. Otherwise, it tries to identify the GOT.PLT section based on the property of GOT.PLT entries, *i.e.*, they either point to instructions in the PLT section or point to functions in other modules.

Supporting ROPscan. Pointer identification collects pointers, especially code pointers, so that the crash report after DESENSITIZATION still contains sufficient information to support the first step of ROPscan – code pointer identification. In the case of the crash generated from Figure 1, all pointers (*i.e.*, `buf`, `string`, `ptr`, G_{pivot}, G_1, G_2, G_3) will be identified and kept in the desensitized crash report.

B. *Bug-Specific and Attack-Specific Modules*

Besides identifying pointers for general purposes, we further investigate other potential data related to program faults and malicious attacks in the crash memory. In particular, DESENSITIZATION considers popular, general offensive methods that are widely used by real-world attacks, and strives to preserve the necessary information in the crash report correspondingly so that developers can diagnose the crash and reveal the attack easily. Other than the supported bugs and attacks, we design DESENSITIZATION to be extensible to support future bug-analysis techniques.

1) Heap Structure: In recent years, heap-based vulnerabilities and exploits have dominated security breaches [6], [41], [10], [31], [32], [64], [43], [23], [77], [7]. These exploits utilize various methods to abuse heap structures that lack sufficient security checks in various implementations. [74] provides a community-wide compilation of well-known heap exploits against `ptmalloc` [34], where each technique corrupts specific heap metadata to achieve certain attack primitives.

To help debug program faults and identify failed attacks involving heap objects, DESENSITIZATION tries to identify and save all heap metadata from the process memory space. Our

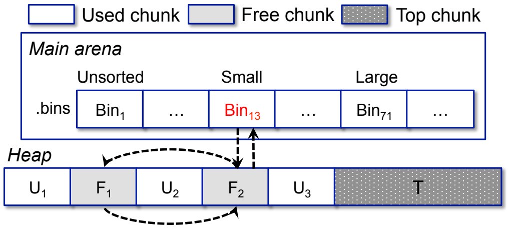


Fig. 3: Heap structures implemented by `ptmalloc` in `glibc`. It embeds metadata, like pointers, into the allocated memory region.

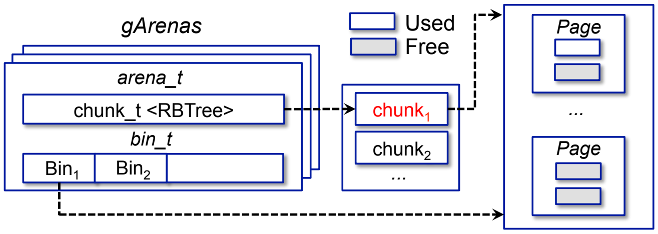


Fig. 4: Heap structures implemented by `jemalloc` in `firefox`. It uses dedicated memory region to store all heap metadata.

heap module is designed to be generic and thus can cover many types of bugs and attacks. For example, from the desensitized crash report, developers can identify corrupted heap structures due to buffer overflows, or they can find dangling pointers of freed memory, which might be the root causes of use-after-free bugs. Currently, our DESENSITIZATION technique supports two common heap allocators, `ptmalloc`, which is used by `glibc` on Linux distributions, and `jemalloc`, which is adopted by `FreeBSD` and other popular applications, such as `firefox`.

Figure 3 presents a typical view of the heap layout in `ptmalloc`, which maintains multiple `arenas` for multi-threaded applications, one for each thread. An `arena` is an instance of structure `malloc_state` and serves the allocation and freeing of its associated thread. Each `arena` contains several large amounts of continuous areas of heap memory. Each heap chunk is in the status of either *allocated* or *freed*, indicated by the chunk metadata. The metadata of an allocated heap chunk contains the `size` field describing the chunk size, the `prev_size` field providing the size of the previous chunk (if it is freed), and several flags bits, like the `PREV_IN_USE` bit indicating whether the previous chunk is freed or not. A freed chunk has two extra metadata fields, the `fd` pointer pointing to the next freed chunk and the `bk` pointer pointing to the previous freed chunk. To support efficient allocation, `ptmalloc` connects freed heap chunks through the `fd` and `bk` pointers, and groups them into different `bins` – one `bin` for all chunks with the same or similar size. To free an allocated chunk, `ptmalloc` inserts it into a proper `bin` according to its size before returning it to the system. To allocate a new chunk, `ptmalloc` tries to find a large enough cached chunk from the `bins` before requesting the memory from the `top` chunk. If the crashed program uses `ptmalloc`, DESENSITIZATION manages to parse the heap metadata as above, starting from the information in arenas that are indicated by the symbol `main_arena`. Then, it walks through all the structures accordingly in an iterative manner, such as tracking each chunk in memory space by its chunk size, until it finds all heap metadata or reaches a parsing error due to corrupted heap information.

Figure 4 presents a typical view of the heap layout in `jemalloc`. `jemalloc` shares some structures with `ptmalloc`, such as `arenas` and `bins`. The main difference is that `jemalloc` saves all metadata in one dedicated memory region, separated from the real payload. This design improves the security of the heap manager, as the common heap overflow cannot reach the metadata as easily as before. It also achieves better performance, as heap structures are maintained as efficient `tree` structures and traversal of these structures is more convenient due to the memory locality. Starting from the main arena (*i.e.*, `gArenas`), DESENSITIZATION parses the tree of arenas (*i.e.*, `struct arena_t`) to fetch all heap structures. While each arena maintains a dynamic list of `bins` (*i.e.*, `struct arena_bin_t`), it also points to another tree of `chunks` (*i.e.*, `struct arena_chunk_t`), which contains a large continuous memory space for allocation, similar to the function of `struct heap_info` for `ptmalloc`. The sizes of the continuous memories are fixed based on native systems (with usually 1 MB = 1 page * 250), and since there is no metadata between user-requested regions anymore, DESENSITIZATION simply records the data in the `struct arena_chunk_t` header, rather than traversing through all user memory like it does for `ptmalloc`. DESENSITIZATION identifies and saves all the structures mentioned above from the process memory.

2) ROP Gadget Chain: Return-oriented programming (ROP) is an indispensable step for modern memory-error exploits [73]. Despite ROP chains tending to be self-destructive during execution, DESENSITIZATION strives to save related information if necessary. Note that the pointers of ROP gadgets have been saved by the pointer identification. Therefore, our ROP module focuses on saving the data in the ROP payload, which can be used as operands of ROP gadgets. Specifically, if a consecutive memory of `K` bytes contains more than `N` code pointers, DESENSITIZATION considers the region as a potential ROP payload and will keep all the data in between, including pointers and non-pointers. The numbers `K` and `N` are tunable to users. In our evaluation, we use `K=48,N=4` for 32-bit systems and `K=96,N=4` for 64-bit systems, based on our experience. Further, DESENSITIZATION conservatively keeps a 96-byte memory region around stack pointer `esp/rsp`, which can provide both a general debugging aid for program faults and tackle those ROP attack-crashing programs.

Supporting ROPscan. ROP gadget chain identification keeps the operands of ROP gadgets inside the desensitized crash report, which supports the second step of ROPscan — the speculative execution and gadget detection. In the example of Figure 1, DESENSITIZATION will keep the operands for G_1, G_2, and G_3. Therefore, ROPscan can start the execution from G_1 and identify three consecutive ROP gadgets.

3) Shellcode: Code injection attacks insert malicious instructions, called *shellcode*, into the memory space of the victim. Although NX and DEP disable direct code injection [3], [29], attackers can create a writable-and-executable data region through `mprotect()` and place shellcode inside, commonly serving as the last exploit step. To search for potential shellcode payloads in a lightweight manner, DESENSITIZATION scans memory dumps for critical system call instructions outside the text section, such as int 80 for x86 platform and syscall for x86_64 platform. For each potential system call instruction, DESENSITIZATION fetches the proceeding 200 bytes and tries to disassemble them. As the fetch might truncate actual payloads, DESENSITIZATION tolerates disassembling errors within the first 16-bytes. If the disassembling process does not find any error after the first 16-bytes, and the assembly code is ended with one system call instruction, we treat the extracted memory region as a valid shellcode payload. To cover shellcode with more than 200 bytes, we repeatedly apply the method to another 200 bytes before the valid shellcode, until we cannot find more shellcode regions. All the identified shellcode payloads are kept by DESENSITIZATION in the desensitized crash report.

4) Malicious Format String: In format string exploits, attackers manipulate printing formats to perform arbitrary read and write with severe consequences. DESENSITIZATION hunts all the valid strings that contain format specifiers defined in the man page of Linux [50] through regular-expression matching, including those with various length and precision fields. DESENSITIZATION simply fetches memory between the two closest null bytes (*i.e.*, 0x00) as an overestimation for each string. Next, it tries to find arguments for printf-like functions. On 32-bit systems such as x86, arguments are passed through stacks while the pointer of the format string is the first argument. DESENSITIZATION searches for pointers that point to any malicious format strings on stacks and saves the six following pointer-size bytes to conservatively keep all arguments. On 64-bit systems such as x86-64, the first several function arguments are passed through registers, where the pointer of the format string will be in register %rdi. In this case, DESENSITIZATION cannot pinpoint the potential locations of corresponding arguments, but just saves malicious format strings and all their references in memory.

C. Supporting Future Analysis

The current design of DESENSITIZATION covers most of the common bugs and attacks, but it is never complete due to various bug types and quickly evolving exploit techniques. Therefore, DESENSITIZATION provides friendly interfaces for researchers to write their own modules to cover more bugs and to support future analysis techniques. It also allows developers to define customized policies based on their domain knowledge, either to aggressively remove more pointers that are known to be unnecessary for debugging or to conservatively keep more non-pointers that are critical for bug analysis. For example, developers can acquire extra data in critical sections to assist backward execution at crash sites [25], [82] or search for more attack payloads in error-prone call frames. DESENSITIZATION provides configurable options for non-developers, such as further removing data pointers in crash reports to avoid uncommon information leakage. By taking various levels of DESENSITIZATION techniques for certain address spaces of the crashed execution, DESENSITIZATION produces new crash reports in a flexible manner.

IV. IMPLEMENTATION

DESENSITIZATION contains two components: the parser for extracting necessary information that needs to be kept in the crash report and the writer that dumps the desensitized data to physical disks in the sparse file format. We implement DESENSITIZATION in 4,284 lines of Python code, publicly available at https://github.com/sslab-gatech/desensitization.

1) Memory Parser: Currently, DESENSITIZATION supports generating desensitized crash reports in coredumps or minidumps. Coredumps contain almost all the memory content of the crashed process, including environment variables, exception information, context of CPU, heap and stack of each thread, loaded libraries, and modules. Minidumps collect much less information to reduce the size of the report. For example, by default, it does not include the heap region of the crashed process. Considering the distinction in formats and design concepts, we implement two different parsers to generate coredumps and minidumps. To generate coredumps and parse binaries and libraries on the Linux platform, DESENSITIZATION relies on an open-source tool [9] with python bindings, pyelftools, for parsing various headers and sections in ELF format and DWARF debugging information. Due to the limited availability of public libraries for generating minidumps in Python, we implement our own minidump parser without relying on third-party codes. For disassembling binaries, DESENSITIZATION works with pyxed [44], the python bindings for Intel X86 Encoder Decoder (XED).

2) Report Writer: To produce minimal crash reports, the writer of DESENSITIZATION stores the desensitized memory as sparse files, as a large number of bytes are cleared to zero. Specifically, it *fseeks* on positions of kept bytes in the processed data to create "holes" in newly generated dumps.

V. EVALUATION

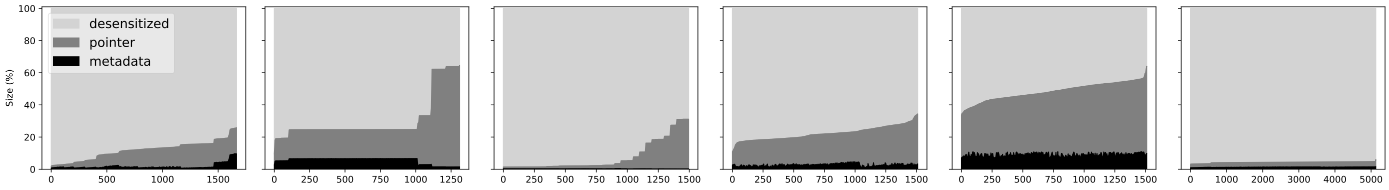
We perform empirical evaluations on real-world crashes to answer the following questions of DESENSITIZATION regarding its effectiveness, usability, and practicality:

- **Privacy protection.** What percentage of bytes can our DESENSITIZATION technique nullify from the process memory? Can DESENSITIZATION remove sensitive data? (§V-B)
- **Attack preservation.** Can DESENSITIZATION retain sufficient information to support common server-side crash analysis? Can DESENSITIZATION preserve attack residue if the program crashes due to failed attacks? (§V-C)
- **Practicality.** How well can DESENSITIZATION help reduce the size of the crash report? Can DESENSITIZATION generate the new crash report in a reasonable time with proper memory usage? (§V-D)

A. Experiment Setup

To compare our new format of crash reports with existing ones, we first generate the reports in old formats using existing tools. Then, we apply our technique on these reports to identify and remove users' sensitive information. Specifically, we rely on the Linux kernel to create coredumps and utilize the breakpad library to generate minidumps. To simplify our description, we denote the crashes triggered by failed attacks as *attacker-driven* crashes (reports) and denote the crashes without any attack involved as *user-driven* crashes (reports).

Generating user-driven reports. We use three different methods to generate user-driven reports: one with the PoCs of real-world bugs, one by fuzzing PoCs, and one by sending



(a) `ffmpeg` **(b)** `php` **(c)** `chakra` **(d)** `ff`-coredump **(e)** `ff`-minidump **(f)** `tachikoma`

Fig. 5: Distribution of non-zero bytes in the original memory space. "desensitized" shows the bytes removed from the memory by DESENSITIZATION, while "pointers" and "metadata" indicate the bytes that remain in the memory. Since the data related to heap structures, ROP gadgets, format strings, and shellcode account for a small portion of the memory, we combine them as "metadata."

abort signals. First, we get 19 PoCs for `ffmpeg`, 15 for `php`, 30 for `chakra`, and 15 for `firefox` from public resources [59], [62], [72], each against a distinct bug. These PoCs simply trigger bugs in programs and immediately crash the process. Second, we utilize the AFL crash mode [83] to keep randomly mutating the benign PoCs for 24 hours to generate more crashes. We believe the random mutation will not introduce malicious actions, and therefore the generated crashes are still user-driven. Third, we use `firefox` to visit the Alex Top 1500 websites and send the `SIGABRT` signal at a random time for the process to terminate abnormally. In general, we obtain 7,507 normal crashes, including 1,667 crashes for `ffmpeg`, 1,313 for `php`, 1,497 for `chakra`, and 3,030 for `firefox`.

Generating attacker-driven reports. We also use three different methods to generate attacker-driven reports. First, we find two exploits against `ffmpeg`, two against `php`, two against `chakra`, and one against `firefox`. The details of the exploited CVEs are discussed in §V-C2. Second, we also utilize the AFL crash mode to collect more crashes. Unlike those user-driven reports generated through fuzzing upon the PoCs without attack payload, the current seed corpus contains fragile, but full chain exploits. Random mutation on the exploits before feeding them to the vulnerable programs makes the executions crash at different exploit stages. We conservatively treat them as attacker-driven crashes without checking each input one by one. Third, we collect crashes from the DEFCON 2015 challenge `tachikoma`. We download the network traffic throughout DEFCON 2015 [28], which presents exploits from top hackers striving to compromise exploitable programs. As the only `x86` binary, `tachikoma` contains 14,966 LoCs with seven intended bugs of various complexity, spanning buffer overflows to logical errors. We extract exploits from 366,792 relevant network sessions and replay them to collect coredumps. Totally, we get 5,883 attacker-driven coredumps, including 400 for `ffmpeg`, 200 for `php`, 100 for `chakra`, 50 for `firefox`, and 5,133 for `tachikoma`. More information on the PoCs, exploits, and crashes can be found in Table VI in §A.

We evaluate DESENSITIZATION on a 32-core machine running Ubuntu 16.04, with Intel Xeon Gold 6136 processors at 3GHz and 128GB memory. We limit DESENSITIZATION to only utilize four cores to process each crash as normal desktops usually have a small number of cores, like four or eight. DESENSITIZATION identifies necessary bug-related and attack-related data from the memory and *nullifies* other bytes to remove potential privacy-sensitive data.

B. *Privacy Protection by* DESENSITIZATION

Since the privacy information is program-specific, it is challenging to quantitatively measure the privacy protection after DESENSITIZATION. Instead, we use three different methods to understand and estimate the privacy protection. First, we measure the reduction of non-zero memory bytes. This reflects the privacy protection when the private information is distributed evenly in the memory. Second, we retrieve the printable strings from the crash reports and measure their reduction during DESENSITIZATION. As much of the common private information is printable strings, this result gives us an estimation of privacy benefits. Third, we manually inspect several cases in which `firefox` leaks the user's private information and confirm that with DESENSITIZATION, such information is successfully nullified.

1) Memory Byte Reduction: Figure 5 shows the distribution of non-zero bytes in the original memory snapshot, grouped by the data types. Each unit on the x-axis represents one crash report, sorted by the percentage of pointers, while the y-axis depicts the percentage of each data type. The `pointer` and the `metadata` are kept in the memory space, while the `desensitized` data are set to zero to protect the user's privacy. The `metadata` data covers identified heap structures, ROP gadget chains, shellcode, and malicious format strings. As they occupy a small portion of all non-zero data, we combine them in the figure. The numeric distribution of the remaining data is given in Table V, where the number is an average.

On average, DESENSITIZATION can remove 80.9% of non-zero bytes from the original memory snapshot; 89.5% of all remaining data are pointers. If the user's privacy is evenly distributed in the memory, we can reduce the possibility of information leakage by 80.9%. In fact, most of the user's private information is allocated in concentrated locations, like a string representing a website cookie. As such information is not likely to be pointers, we believe the real result of privacy protection should be more appealing than what we reported here.

Figure 5a shows the result of applying DESENSITIZATION on `ffmpeg` coredumps. On average, DESENSITIZATION can nullify 93.50% of all non-zero bytes for each coredump. Among the remaining data, 77% of them are pointers. Heap metadata and ROP payloads account for 9.83% and 13.0%, respectively. Shellcode payloads and malicious strings take negligible portions in the desensitized memory. We inspect the memory snapshot manually and find that a large portion of the original memory contains multimedia files or part of their

Prog	Size (K)	Pointers	Heap	ROP	Shcode	FmtStr
ffmpeg	366	77.0%	9.83%	13.0%	.110%	.004%
php	974	90.2%	8.45%	1.28%	.043%	.016%
chakra	1,567	97.2%	1.04%	1.74%	.014%	0
ff-core	9,953	88.7%	2.95%	8.25%	.017%	.052%
ff-mini	6	82.4%	0	17.6%	0	0
tachikoma	12	68.6%	2.02%	27.8%	1.59%	0
avg.	-	89.5%	3.33%	7.09%	0.03%	0.04%

TABLE V: Average distribution of remaining data per crash report produced by Desensitization across evaluating benchmarks.

copies. These data usually do not affect the bug analysis but may reveal the user's watching history. Removing them will protect user privacy.

Figure 5b shows the result of desensitizing php coredumps. Similarly, Desensitization can nullify around 58.9% of non-zero data in each crashed memory and identify 37.1% of useful data as pointers and 4.03% as other metadata. However, for php, the ratios of useful data vary significantly from crash to crash: in the worst case, 64.9% of data is kept in the desensitized memory, while in the best case, less than 10% of non-zero data will remain. We inspect the crashes and find that as a language interpreter, php provides many functionalities, like parsing multimedia files and maintaining network interfaces. The collected PoCs may take different execution paths and crash the program in diverse locations, resulting in different ratios of useful data. For example, one PoC crashes the program by triggering an infinite recursive call, leading to a large number of stack frames. As each stack frame has many data pointers, like return addresses and saved esp/rsp, Desensitization only nullifies 35.1% of all non-zero data

The evaluation result of chakra in Figure 5c suggests that Desensitization can consistently nullify about 89.1% of all non-zero bytes in each coredump. Meanwhile, an average of 10.6%, 0.11%, and 0.19% of the original data are identified and thus kept as pointers, heap structures, and potential ROP residues. Other metadata are negligible (less than 0.002%). We have to clarify that chakra relies on native systems for memory allocation but maintains its own customized metadata. Currently, Desensitization is only able to identify the relevant heap structures from *ptmalloc*, and corruption on custom heap metadata will be missed during the attack analysis. We can extend our heap module to support the customized metadata of chakra, as we have done for ptmalloc and jemalloc. We leave this as a future work.

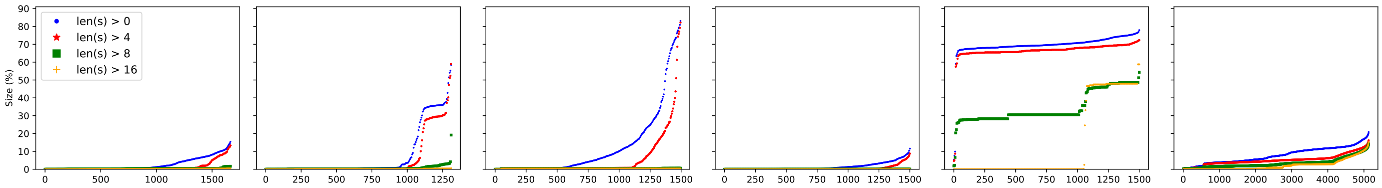
Figure 5d and Figure 5e provide the results of applying Desensitization on firefox crashes, the former for coredumps and the latter for minidumps. Figure 5d shows that Desensitization can nullify 77.8% of non-zero data in coredumps and leave 19.7% as pointers, 0.65% as heap structures, and 1.83% as potential ROP residues. Figure 5e indicates that Desensitization maintains 42.0% of non-zero bytes as pointers (with no heaps) and 9.00% as ROP residues, and nullifies 49.0% from each original memory. Apparently, Desensitization identifies more portions of pointers on minidumps. The reason is that a majority of non-pointer data of firefox resides in the heap region, which are not captured by minidumps. Our result indicates that minidump is less effective for developers to diagnose bugs or detect attacks. Nevertheless, our Desensitization can still remove a large amount of unnecessary data from minidumps to protect the user's privacy.

Figure 5f presents the results of Desensitization on crashes of tachikoma. In particular, Desensitization can nullify on average 95.6% of non-zero bytes in the coredumps. While most are removed, Desensitization manages to keep 3.02% of the original memory as pointers, 0.09% as heap structures, and 1.23% as potential ROP residues. Since the coredumps of tachikoma are produced by failed attacks against the synthesized binary, they are not comparable in terms of sizes and complexity (*e.g.*, heap structures) to those from previous benchmarks. However, Desensitization is shown to be consistent in removing data for special-purpose programs.

2) Reducing Printable Strings: To further quantify the privacy benefit, we measured the reduction of printable strings by Desensitization from the original reports. We define a *printable string* as a sequence of printable characters with a minimum length N. In our evaluation, we use different values of N, including 1, 5, 9, and 17. Overall, the evaluation serves as a proof-of-concept for illustration due to the lack of common schemes for measuring privacy leakage. Figure 6 presents the evaluation results for all benchmarks with different definitions of printable strings. Each figure shows the percentage of printable strings *left* in the desensitized reports. In general, Desensitization can remove more than 95.0% of printable bytes from the coredumps of ffmpeg, php, firefox, and tachikoma, achieving a significant privacy benefit. Further, if we define printable strings to have at least 9 bytes, Desensitization achieves almost 100% of string reduction for all reports from all programs. For those bytes that cannot be effectively removed, we found that they are mainly kept by the modules of ROP, format string, and shellcode as suspicious payloads or arguments.

For chakra coredumps, Desensitization manages to remove 84.2% of printable strings with the conservative definition, *i.e.*, N=1. However, as shown in Figure 6c, in extreme cases, 83.7% of such bytes are left in the reports. Through manual examination, we learned that those are actually false positives due to the special address layout in certain executions. Particularly, most printable strings left by Desensitization have a short length from 2 to 6 bytes. While these bytes look like printable strings, they are in fact pointers, such as 0x55554730 ("0GUU"). If we change the definition of printable string to require at least nine consecutive printable bytes (per alignment of x86-64 systems), Desensitization can remove almost all of them. The same explanation applies to certain cases of php coredumps in Figure 6b as well.

Figure 6e shows that Desensitization is less effective when handling the minidumps of firefox. With the conservative definition, it only removes 37.0% of printable bytes on average. Even if we increase the requirement of the length, Desensitization still leaves more than half of printable strings in the desensitized reports. This is mainly because most user data, especially printable strings, are typically stored in the non-stack regions and thus are not saved in minidumps when the program crashes. With a limited number of printable bytes, even a few false positives (like those we explain above) will hurt the statistics of Desensitization. For example, when handling strings with at least 9 bytes, Desensitization can



(a) ffmpeg **(b)** php **(c)** chakra **(d)** ff-coredump **(e)** ff-minidump **(f)** tachikoma

Fig. 6: Percentages of printable strings left in desensitized crash reports. We performed evaluation with different definitions (lengths) of strings, shown in different colors. The overall result is that long strings are less common than short strings.

remove 154 bytes on average, leaving only 52 bytes from each original crash. Yet the result still gives the reduction ratio of printable bytes as 74.8%. Though DESENSITIZATION is not designed to be perfect in terms of privacy preserving, Figure 6e can be misleading in this sense.

We also performed an automated scan of private information with simple regular expressions, followed by a manual analysis of the scanning results. We found several categories of interesting information left in the original crash reports but removed by DESENSITIZATION. The information includes but is not limited to local directories, local environment variables, usernames, passwords, cookies, visiting URLs, source IPs, network protocols, geo-location coordinates, and snippets of user scripts yet to be executed. §V-B3 provides selective case studies in more detail for `firefox`.

3) Case Studies on Privacy Protection: We studied the source code and the runtime of `firefox` and identified three scenarios where a minidump of the program will leak the user's privacy, such as website cookies and user inputs. We inspected the desensitized minidumps and confirmed that DESENSITIZATION successfully removes all these private data.

```
// Subclass of nsTString that allocates stack-based string.
class nsTAutoStringN : public nsTString<T> {
  public:
    nsTAutoStringN() : string_type(mStorage, 0, ...),
                       mInlineCapacity(N - 1) {
      mStorage[0] = char_type(0);  // null-terminate
    } ...
    static const size_t kStorageSize = N;
  protected:
    friend class nsTSubstring<T>;
    size_type mInlineCapacity;
  private:
    char_type mStorage[N];     // internal buffer on stack
}
```

Fig. 7: Definition of `nsTAutoStringN` in `firefox`, inherited by `nsAutoString` and `nsAutoCString`. The internal buffer `mStorage` may leak users' private information (shaded line).

Source of Leakage. `firefox` adopts many techniques to make the software work efficiently. One of these techniques is the implementation of auto string classes, namely, `nsTAutoStringN`, as shown in Figure 7. By default, the payload of a string is allocated dynamically from the heap region, while the class only keeps some structural information, including a pointer pointing to the real payload in the heap. However, the class also contains a fixed-size array, called `mStorage`, and uses it to store the string content if the string is shorter than the array. This design choice makes access to the string content more efficient due to the memory locality and meanwhile renders the object management simpler, like no explicit `malloc` or `free` of the payload. The `nsTAutoStringN` class is inherited by many others, such as `nsAutoString` and `nsAutoCString`. While the former is designed for wide characters with an internal buffer of 128 bytes, the latter has 64 bytes for narrow characters. If a small string of these classes is allocated on the stack, its content will be on the stack as well. Since the minidump keeps all stack content of the crashing thread, short strings will be dumped into the minidump, leading to privacy issues.

Leaking Cookies. The design of `firefox` adopts the principle of privilege isolation. In particular, the parent process does all the security-critical work, like file access and network connection establishment, while child processes merely serve for rendering and parsing. In function `SetCookiStringInternal()` of Figure 13 in §B, a child process is in charge of parsing cookies and storing cookie attributes (*e.g.*, name, value, host and expiring time) into stack variables of class `nsCookieAttributes`. As class `nsCookieAttributes` is derived from class `nsAutoCString`, any cookie attribute shorter than 64 bytes is stored in its internal buffer, which is in turn allocated on the stack. Therefore, any crashes inside this function will dump the cookie information to the crash report as long as the cookie is shorter than 64 bytes. We inject a bug in `firefox` to make the execution crash inside function `SetCookiStringInternal()` and use the modified `firefox` to visit popular websites. Most of them will crash and generate a minidump file, where we can find the cookie information of the corresponding website. DESENSITIZATION is able to nullify the cookie contents, as they are unlikely to be pointers, heap metadata, or any other attack residues.

Leaking User's Inputs through Form Submission. Form submission is a common way for users to interact with web applications, like submitting user names and passwords for authentication. Figure 14 in §B shows the code of `firefox` used for preparing the form data as requested by users. In memory, each form field is always in its plaintext unless the developer specifies one of multiple `enctype` attributes to it, as defined in function `GetFromForm()`. When `firefox` handles forms of `multipart/form-data` for users to upload files, function `FSMultipartFormData::AddNameValuePair()` places user inputs in objects of class `nsAutoCString`, which are allocated on the stack in plaintext. Further, many *hidden* form fields collect user information for web tracking, like geographic locations and timestamps. All of them may leak users' private information through minidumps. We use the same method as before to verify that the minidump contains user input if the execution crashes inside function `FSMultipartFormData::AddNameValuePair()`. Depending on the concrete website, we find extremely sensitive information from form fields on the stack, like user names, passwords, or even credit card information. Desensitization helps eliminate these sensitive data from minidump files, as most of them are alphanumeric characters, which are not confused with other debug information in the crash report.

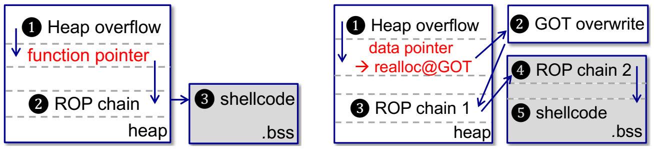
Leaking Filling History through Autofilling. Autofilling is a convenient feature for users to fill a form quickly from historical inputs. Specifically, when an HTML textbox (*e.g.*, `<form>`, `<input>`, `textarea`) has the `autocomplete` attribute, the browser will provide users a list of historical inputs. Figure 15 in §B shows that in function `EnterMatch()`, a single user-provided character triggers the browser to find matched historical inputs and shows the results in a drop-down list. When users navigate the drop-down list with up/down keys, or they select one record from the list, the highlighted or selected record will be stored in a stack variable of class `nsAutoString`. If the execution crashes here, the user information will be copied into the minidump. Note that some websites use customized JavaScript to handle the autofilling by specifying the action type of forms, where the filling history may be used differently. Similarly, we verify that the minidump contains the filling history if `firefox` crashes at particular functions. We confirm that Desensitization clears the sensitive information from the collected minidumps.

C. Supporting Bug-analysis and Attack-detection

We evaluate desensitized crash reports to verify that they still contain sufficient information for popular bug-diagnosis and attack-analysis techniques. Our evaluation is performed in two ways. First, we feed the crash reports before and after Desensitization to existing bug-diagnosis tools and expect the same analysis results. Second, we inspect several attacker-driven crashes to make sure that all data related to the attack construction are kept in the desensitized crash reports.

1) Supporting Bug Analysis: We use `Socorro` [57] and `Backtrace` [8] as two bug-analysis tools for our evaluation. The former is the default crash reporting service used by Mozilla products, while the latter is a more generic commercial software recommended by Mozilla for providing signature-based crash classification. Both tools rely heavily on call stacks in crash reports along with additional heuristics to generate a signature for each crash. Crash reports with the same signature are grouped together, and the number of crash reports in each group is used to determine the priority for detailed analysis.

We feed all the collected coredumps before and after Desensitization to `Backtrace` and compare their crash signatures. We also submit the minidumps of `firefox` to `Socorro` through its native crash reporting service and compare signatures per crash. The result shows that Desensitization has no impact on the crash signatures of any dumps, including coredumps of `ffmpeg`, `php`, `chakra`, and `firefox`, as well as minidumps of `firefox`. In other words, the crash signatures generated by either tool are always the same before and after our Desensitization on each crash report. This findings validate that Desensitization merely removes unnecessary data safely from the crash reports without altering any useful information, at least for debugging processes like classification.



(a) CVE-2016-10190 **(b)** CVE-2016-10191

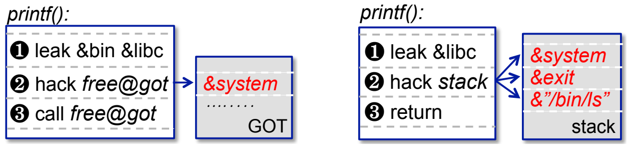
Fig. 8: Exploit steps against `ffmpeg` via two CVEs. Both CVEs are heap overflows, and both attacks utilize ROP and code injection.

2) Case Studies of Attack Preservation: We illustrate how the information identified and saved in crash reports by Desensitization, such as pointers and heap metadata, can support real-world attack analyses. Particularly, we implemented several proof-of-concept tools to detect attack residues from crash reports before and after Desensitization. The first tool detects the corruption of various heap structures to identify heap-based vulnerabilities. It walks through the linked lists to find inconsistent metadata, such as abnormal chunk size, broken linked bins, and dangling pointers. The second tool searches for malicious format strings to detect format string attacks. It feeds each potentially malicious string to a `printf`-like function with other identified arguments and verifies that the function's side-effect to memory is consistent with the crash report. The third tool scans for ROP residues left in the crash report. For each potential ROP payload, the tool runs the gadgets through instruction emulation and verifies that the side-effects to stack and frame register (*i.e.*, `esp` or `rsp`) match the crash report. The last tool validates GOT.PLT entries with the help of external debug symbols. We develop these tools based on our experience of attack analysis for the purpose of verification. Developers can pick other techniques to analyze a crash report [21], [67], [37], [75], [66], [65].

As we mention in §V-A, we feed collected exploits to each vulnerable program and collect the resulting crashes. Meanwhile, we collect coredumps from `tachikoma` by replaying the exploit traffic. Then, we apply our PoC tools against the original crash reports and the desensitized ones. Our tools report the same result about the failed attack for both versions of the crash reports, showing that Desensitization keeps important attack residues in the desensitized report.

ffmpeg. Figure 8 depicts exploits against two heap overflow bugs of `ffmpeg`: CVE-2016-10190 and CVE-2016-10191. Both attacks corrupt pointers on heap and stitch ROP gadgets and finally execute malicious shellcode, but they differ slightly in terms of corruption targets and gadget chaining due to the nature of the vulnerabilities. In general, we collect 400 coredumps from the failing exploits through fuzzing. On average, Desensitization is able to nullify 93.5% of irrelevant data as described in §V-B and keep those of pointers and metadata. Among all remaining bytes, 47,701 bytes are ROP payloads; 386 bytes are potential shellcode payload; 42,200 bytes are identified as GOT.PLT entries. Although format string attacks are not involved, Desensitization saves another 13 bytes regarding format strings and related arguments.

Since Desensitization maintains the heap metadata, our analysis tool on heap integrity is able to detect the same overflown heap chunks in the desensitized crash report as those



(a) CVE-2015-8617 **(b)** CVE-2016-4071

Fig. 9: Exploit steps against php via two CVEs. Both CVEs are format string bugs, and both attacks use the ret2libc technique.

in the original one. For exploits that crash in the ROP chains, Desensitization maintains the memory values pointed by stack pointers and keeps all the suspicious gadgets in the crash report, helping to capture the residual ROP gadgets. For those overwriting GOT.PLT entries with incorrect addresses, Desensitization identifies entries in the memory and keeps them in the crash report for validation. When the program crashes in shellcode, most payloads are still in the crash report, as Desensitization conservatively retains all potential ones.

php. Figure 9 shows exploits against two format string vulnerabilities of `php`: CVE-2015-8617 and CVE-2016-4071. The former allows attackers to learn the randomized function address, while the latter recursively modifies the `ebp` values on the stack until it overwrites the return address outside the vulnerable function. We gather 200 coredumps from the failing exploits through fuzzing. The proportions of unnecessary data, pointers, and other metadata are similar to those of previous ones. Particularly, Desensitization collects around 154 bytes of malicious format strings along with their arguments. Desensitization additionally saves 12,494 bytes for potential ROP payloads and 417 bytes for shellcode, which are apparently false negatives. Meanwhile, Desensitization identifies 6,632 bytes for GOT.PLT entries.

Desensitization conservatively saves potential format strings and their relevant pointer arguments in crash reports, and allows our analysis tool to track down the attacks with the desensitized crash reports in most cases. Note that 32-bit systems are easier to tackle, as arguments are stored in the stack region. Therefore, Desensitization can find the format string pointer and keep their pointer arguments in relative offset based on the format specifiers, even if the arguments are non-pointer data in randomly mutated exploits. However, since 64-bit systems pass first several arguments through registers, it is hard for Desensitization to identify possible format string arguments. Thus, Desensitization only keeps pointers as designed and might remove some arguments when they are non-pointer data, resulting in false negatives. As `php` is compiled as a 32-bit binary, Desensitization can save necessary information about format strings to support our analysis.

chakra. We select CVE-2016-0193 and CVE-2017-0266 of `chakra` for crash generation, which are caused by heap overrun and type confusion due to incorrect JIT optimization. The exploit of CVE-2016-0193 corrupts the length field of an array object on the heap and uses the array to overwrite a virtual table pointer to hijack the control flow for ROP attacks. CVE-2017-0266 allows attackers to craft a fake `DataView` object on the user-controlled memory for arbitrary memory read and write, thereby corrupting GOT.PLT entries to execute arbitrary code. In sum, we collect 100 coredumps from failed exploits. Desensitization successfully removes the irrelevant data while keeping pointers and heap structures in a ratio similar to others. Furthermore, Desensitization saves around 221 bytes of shellcode payloads and 27,291 bytes of ROP residues, as well as 10,072 bytes of GOT.PLT entries, but fails to find any malicious format string in this case. Although `chakra` uses a customized heap manager, Desensitization manages to save pointers that are relevant to ROP chain residues, as well as identified GOT.PLT entries in the crash report.

firefox. We generate `firefox` crashes by exploiting CVE-2019-9810, which is used in Pwn2own 2019 [60]. The bug is an array out-of-bound access due to an incorrect JIT optimization. Our exploit leverages the bug to corrupt the length field of an `Uint32Array` object on the heap, which helps to bypass ASLR and leak the base address of module `libxul.so`. Furthermore, we use the corrupted `Uint32Array` object to modify the buffer address of an adjacent `Uint32Array` object to achieve arbitrary memory read and write. The exploit controls the `PC` value by corrupting the virtual table of a `HTMLDivElement` object with the address of a crafted virtual table. The stack is then pivoted to the heap afterward, where an ROP chain is to be executed. The ROP payload changes the permission of a particular page as both writable and executable and executes the shellcode there. By random mutation through a fuzzing-based approach, we gather 50 coredumps of `firefox` from failed exploits. As usual, Desensitization collects pointers and heap structures while nullifying unnecessary data. In addition, it saves 1,718 bytes as shellcode and 5,129 bytes as malicious format strings. Although our tool based on `ptmalloc` does not apply in this case, Desensitization maintains 820,904 bytes for ROP residues and 9,352 bytes for GOT.PLT entries in the crash report, which will help other tools on ROP and shellcode detection to pinpoint the attack details [21], [67], [37], [75], [66], [65].

tachikoma. The challenge `tachikoma` emulates an interactive war game and introduces seven vulnerabilities for attackers to exploit it. Most of the crashes do not have attack residues as much network traffic is just random bytes for sniffing. Our analysis tools consistently pick up two attack vectors against distinct vulnerabilities before and after applying Desensitization. The first is caused by attackers overflowing a fixed-size stack buffer by copying user input from the heap, but it fails due to miscalculated offsets. Our tools can detect the misaligned heap chunks and the ROP-like residues that end up crashing the binary. The second is caused by attackers exploiting another heap overflow vulnerability to redirect the control flow to the shellcode on the heap. Accordingly, our tools can catch the heap violation, stack pivoting, and prepared shellcode. Our results coincide with the statistics of the CTF, where only two to three of the seven vulnerabilities have been found and exploited by the attending teams. Meanwhile, Desensitization is able to remove the irrelevant data while keeping 8,503 bytes of pointers and 288 bytes for GOT.PLT entries. It saves 251 bytes as heap structures, 3,447 bytes as ROP residues, and 197 bytes as shellcode, successfully maintaining useful information for our analysis tools to pinpoint exploits.

D. Practicality of Desensitization

1) Performance of Desensitization*:* Figure 10 shows the performance of Desensitization on desensitizing crashes

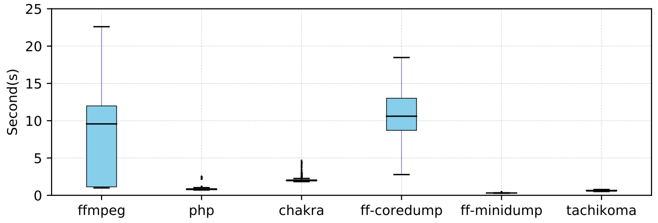


Fig. 10: Time for generating crash report by DESENSITIZATION. Mostly, it incurs reasonable time of less than 15 seconds.

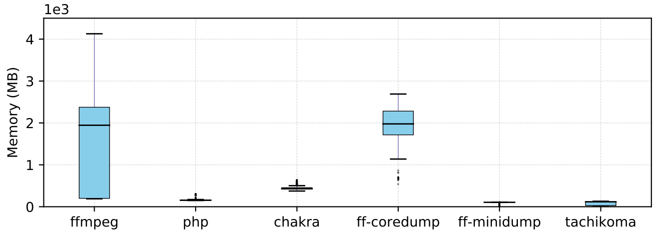


Fig. 11: Peak memory usage for desensitizing crash reports. Mostly, it incurs reasonable memory usage of less than 2.5 gigabytes.

across evaluating benchmarks. In general, DESENSITIZATION takes less than 15 seconds to process each program crash, including the steps of identifying useful data in different types, nullifying other data bytes, and writing the memory as a sparse file. While coredumps from `ffmpeg` and `firefox` have a median of 9 and 11 seconds respectively, DESENSITIZATION takes less than 2 seconds consistently for those of `php`, `chakra` and `tachikoma`. The time needed to process each minidump (*i.e.*, 0.5s) is significantly less than that of analyzing coredump (*i.e.*, 15s) due to their intuitive size differences.

Our optimizations on DESENSITIZATION can significantly reduce the processing time. First, we find that loading a coredump file into memory can result in an order of magnitude more memory usage than the file size. For example, a coredump file of `ffmpeg` is about 150 MB on disk, but it is bloated to 1.8 GB when loaded into memory. The extra memory space, where our analysis should ignore, contains only null bytes. We optimize this process by only loading each program segment using its `file_size` instead of `memory_size`. In this way, the time required by DESENSITIZATION is reduced by 56%. Second, the number of mapped memory regions also affects the overhead of DESENSITIZATION. Specifically, crashes from `firefox` can include up to 340 mapped memory ranges, causing a heavy burden to DESENSITIZATION for validating pointers. Therefore, we merge consecutive memory regions to reduce the number of comparisons. Furthermore, we maintain the merged memory regions as an interval tree, sorting and searching address ranges in logarithmic time complexity. The overhead is therefore reduced by another 77%.

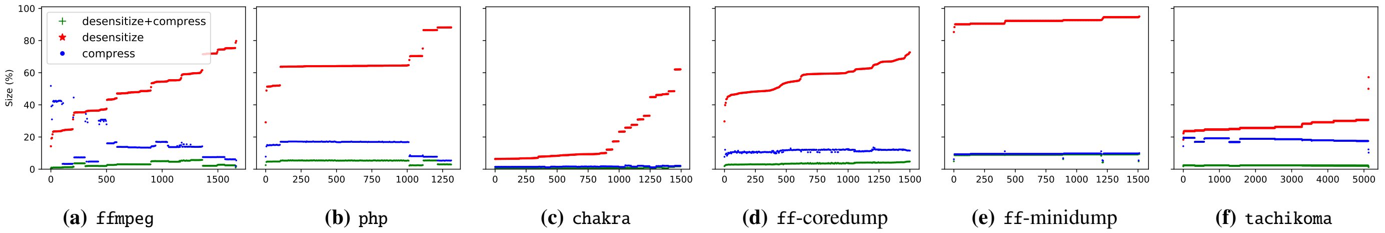
Figure 11 shows the peak memory usage for processing crashes across benchmarks. In general, DESENSITIZATION consumes less than 2.5 gigabytes at the peak to process each crash. While coredumps from `ffmpeg` and `firefox` incur higher memory usage with medians of 2 gigabytes, those of `php`, `chakra` and `tachikoma` only takes around 154, 429 and 117 megabytes per crash, respectively. Again, DESENSITIZATION needs less memory resource to process each minidump (*i.e.*, 104 MB). In general, the memory usage is consistent with the file sizes of processed crash reports. As we mentioned above, most coredumps from `ffmpeg` and `firefox` present physical sizes of more than 1 gigabytes, and so simply reformatting them into their memory representation requires a considerable amount of memory usage. Meanwhile, the complexity of structures within the crash reports, such as the heap regions, also contributes to the finding as DESENSITIZATION needs more memory to preserve those pieces of information.

2) File Size Reduction: As DESENSITIZATION strives to nullify potential sensitive data, it creates a significant number of zero bytes in the desensitized memory. By treating consecutive zero bytes as *holes*, we can save the crash report – in a storage-economy and bandwidth-economy way – as a *sparse* file, which only keeps non-zero bytes. Most modern filesystems, like `ext4` on Linux and `NTFS` on Windows, support sparse files. DESENSITIZATION also facilitates compression algorithms, which adopt similar concepts. To understand the advantage of DESENSITIZATION on file storage, we generate crash reports in four settings: 1) storing the original memory as a sparse file, denoted as *orig-sparse*; 2) storing the desensitized memory as a sparse file, denoted as *desen-sparse*; 3) compressing the orig-sparse file using the `7zip` program, denoted as *comp-sparse*; 4) compressing the desen-sparse file using `7zip`, denoted as *comp-desen*. Figure 12 shows the size of different crash reports, normalized to the size of the orig-sparse file. The crashes on the x-axis are sorted by the percentage of identified pointers.

The figures in Figure 12 (except Figure 12e) show that DESENSITIZATION can significantly reduce the file size of coredumps, regardless of whether they are compressed or not. On average, the sparse file of the desensitized memory is only 49.5% of the original sparse file. With compression, a comp-desen file is only 28.7% of the comp-sparse file. For the `chakra` program (Figure 12c), DESENSITIZATION achieves the maximum reduction of crash reports, where the desen-sparse file only takes 26.1% of the original size, and the comp-desen file saves 99.3% of disk space. In the worst case, DESENSITIZATION can save 24.7% of storage for the `php` program. When using compression, it still reduces the report size to 4.48%.

From these result we can see that the effectiveness of DESENSITIZATION on reducing report size has a strong negative correlation with the density of pointers (each figure is sorted by the percentage of pointers). This result is reasonable because, as shown in Table V, pointers dominate the remaining non-zero data, on average 89.5%. Therefore, a memory space with a low ratio of pointers will be significantly desensitized and left with many large holes of null bytes, leading to a high reduction rate by sparse files. However, the compression ratio is less related to the percentage of pointers. Since the compression mainly depends on the frequency and diversity of data, pointer density is less important in this case. Although the results of `ffmpeg` and `tachikoma` suggest a positive correlation between them, we cannot find a similar relationship from other figures.

An interesting observation from Figure 12a is that when the percentage of pointers is small, the desen-sparse file



(a) `ffmpeg` **(b)** `php` **(c)** `chakra` **(d)** `ff-coredump` **(e)** `ff-minidump` **(f)** `tachikoma`

Fig. 12: Crash report size with various reduction techniques, unified to the sparse file of the original memory. `desensitize` saves the desensitized memory as sparse file; `compress` compresses the sparse file of the original memory; `desensitize+compress` compresses the sparse file of the desensitized memory.

has a smaller size than the comp-sparse file. In these cases, Desensitization is more effective than compression on reducing file sizes. By manually examining the original memory, we find that those crashes take large multimedia files as inputs, which are left in memory when the program crashes. As the input data vary significantly in terms of frequencies and patterns, the compression can only reduce them to around 40% of the original file. However, Desensitization manages to remove most of the input data so that holes of null bytes dominate the desensitized memory, making the sparse file more effective than compression in reducing the file size.

A comparison between Figure 12d and Figure 12e reveals that Desensitization reduces file size more effectively for coredumps than for minidumps, while the compression works well for both. Specifically, the desen-sparse files are only 55.8% of the original sparse files for coredumps, while for minidumps, the number is around 92.3%. Simply compressing the original spare files results in 11.2% of the original size for coredumps, and for minidumps, the number is even smaller, 9.29%. The ineffectiveness against minidumps by Desensitization is caused by the selective dumping of the crashed memory. As minidumps mainly contain thread stacks where many pointers exist, Desensitization cannot desensitize as much as that for the coredumps. Though Desensitization manages to nullify 49.0% of non-zero bytes in minidumps, the removed bytes are usually inter-mixed with other untouched ones on the stack, rendering the sparse file less effective in terms of size reduction. Since the stack pointers are similar to each other, especially for the saved `esp/rsp`, the compression method is still able to significantly reduce the file size.

Figure 12f shows that Desensitization can reduce the sizes of `tachikoma` coredumps in a consistent ratio. The result is consistent with the findings on memory byte reduction in Figure 5f, where Desensitization can remove data in a similar percentage for all crashes. In some extreme cases, the desen-sparse files after Desensitization are still as high as 67.44% of the orig-sparse files. By manually examining these crashes, we find that attackers tried to aggressively spray the heap region with consecutive identical objects filled by data pointers to increase the chance of a successful exploitation. As Desensitization keeps all pointers, the desensitized sparse file cannot achieve a great reduced size. However, it is necessary to keep all such pointers to assist server-side attack analysis. Meanwhile, the comp-sparse file shows a better compression ratio than most others for this case due to the consecutive identical objects with repeated patterns.

VI. Discussion

We discuss several important aspects of Desensitization, including its support to sophisticated attack analysis techniques and possible attacks against the new crash report format.

A. *Analyzability of Desensitized Crashes*

Desensitization strives to achieve a new balance between user privacy and the analyzability of the crash reports. Our design of Desensitization follows a common practice of protecting user privacy that achieves a significant privacy benefit at the cost of sacrificing some functionalities. For example, the widely adopted minidump aggressively drops all data of the crash outside the stack, with the observation that stack data plays an important role in triaging and analyzing bugs [1]. Meanwhile, our tool starts from known analysis techniques and aims to shrink the crash while keeping necessary information.

To mitigate the reduction of analyzability and support special use cases, we design Desensitization in a modular mode so that developers can customize it to compensate for its security guarantee. We believe any practical solutions for a large, complex piece of software would require domain knowledge from experts, such as the developers of real-world applications. We provide interfaces to allow developers to specify crash memories for complete collection or removal of data within the ranges. For example, data near the stack pointers at crash sites can be saved if needed, which might be useful for limited reverse execution by developers. On the other hand, matched data patterns through regular expression can also be selectively preserved when specified.

B. *Hybrid Bug-report Model*

Crash reports from a large number of users are highly redundant: many crashes are caused by the same bug. Therefore, current crash report systems widely adopt automatic bug-classification techniques to group crash reports based on their root causes and prioritize bugs with higher frequencies [57], [8]. In this case, our Desensitization technique helps reduce a large amount of network bandwidth and server-side storage, as it removes crash-unrelated information. However, since developers can benefit from a detailed memory snapshot, or even the crashing input, we can integrate our method into a hybrid bug-report system. Specifically, for most end-users, we deploy Desensitization to protect user privacy and reduce the crash report size. Meanwhile, we can have some volunteer users submit a complete memory snapshot without DESENSITIZATION to assist developers in debugging the program. To protect the privacy of volunteer users, the crash report system can randomly select some from all volunteer users and submit their complete crash report to developers. This hybrid model has less bandwidth and storage while providing sufficient information for debugging.

C. Attacks against DESENSITIZATION

Although DESENSITIZATION mainly adopts systematic approaches for preserving necessary debugging information, some design choices such as identifying thresholds for attacks are considered heuristic. While most are decided based on the survey of exploits publicly available, such as the parameters for identifying ROP and shellcode payloads, the numbers can be ad-hoc and thus provide a limited security guarantee. A stealthy attacker, for example, can craft ROP gadgets with long ROP sequences to bypass the detection from DESENSITIZATION, so that the resulting attack residues would be partially removed. Nevertheless, all the pointers involved in the gadget chains would still be preserved in this case, leaving the possibility for developers to track them down. Meanwhile, as discussed above, DESENSITIZATION offers interfaces for customization, including the preserving rules for crashing data. Therefore, more sophisticated identification techniques from existing works [42], [66], [65], [21], [37] can easily co-exist with the current prototype once specified by the developers, with a potential trade-off between accuracy and performance.

VII. RELATED WORK

SCRASH removes sensitive data from crash files to protect users' privacy [13]. It relies on developers to manually annotate sensitive variables and uses data-flow analysis to identify more sensitive data. It allocates sensitive data in a separated memory region and quickly erases them during program crashes. However, manual annotation incurs a heavy burden for developers and the data-flow analysis has limited scalability; the analysis requires program source code and thus does not work on legacy binaries. Our system automatically identifies and removes sensitive data through crash file analysis and thus does not affect normal executions.

Brickell *et al*. propose a solution [12] to address the privacy issue when multiple users share their bug reports [38]. In this case, a server accepts many bug reports from professional users. Then, the server adopts a machine-learning algorithm to train a model to assign the description to each new, ambiguous bug report from normal users. During the training and testing, the server does not want the user to learn the internal of the trained model, while the user does not want the server to learn her execution traces. The proposed method combines several cryptographic techniques to achieve its goal. Instead of sharing the full bug report, our system removes sensitive information from the crash file and minimizes the chance to leak a user's private information. We can use their cryptographic techniques to protect the data left in the crash file, but that will require a significant change on the server side.

Castro *et al*. try to generate a different input that forces the program to execute (*i.e.*, crash) in the same path as the crashing input and send the new input to the server [19]. PANALYST moves most of the work from the client to the server and relies on client-server interaction to construct the new input [80]. To achieve this goal, both systems rely on a logging system to record the crashing input, a tracing tool to collect the execution path, a taint analysis engine to track the user input, a symbolic execution engine to capture the path constraints, and an SMT solver to provide a new input. These techniques are computation-intensive and require a lot of dependencies, either on the client side or on the server side, and thus provide very limited deployability. CAMOUFLAGE embeds two optimizations to reduce the information leakage from the new input, specifically, path condition relaxation, which modifies the path constraints to allow more satisfying inputs, and breakable input conditions, which prevent the new input from sharing bytes with the original one [24]. RETOME improves the efficiency of symbolic execution by focusing on input-related branches [4]. Our system identifies and removes debugging-irrelevant data from the crash file locally and thus is self-contained and efficient.

Budi *et al*. propose kb-anonymity to anonymize sensitive datasets while preserving their behaviors before sending them out for program testing and debugging [14]. The work focuses on achieving indistinguishable property between multiple homogeneous data items – the sensitive data is retained but cannot be linked to individuals. Our work cleans up personal sensitive data for each crash file, which guarantees the anonymity.

A different input triggering the same execution path as the original one may still leak some private information, especially the path constraints that contain strict requirement on the inputs. To further protect user privacy, several works search for alternative execution paths that have relaxed constraints on inputs while triggering the same crash [52], [53], [54]. MPP relies on symbolic execution to explore program states to find all paths that induce the same crash [52]. REAP and RESPA improve the practicability of MPP by limiting the scope of symbolic execution [53], [54]. Specifically, REAP places randomized, bounded detours around the original path to avoid the path explosion issue. RESPA takes a principled way to retain necessary nodes and find the shortest path to reach the buggy location with the least information leakage.

Satvat *et al*. perform an empirical study on 2.5 million crash reports and find an enormous amount of sensitive information, like token IDs, session IDs, and passwords [69]. This result confirms that current common crash report systems are leaking users' privacy to software vendors. The authors propose a simple pattern-based method to identify and replace sensitive data in URLs and bug descriptions. Our system focuses on the crash file, which may contain more sensitive information.

VIII. CONCLUSION

We propose DESENSITIZATION, an extensible framework that generates privacy-aware and attack-preserving crash reports. DESENSITIZATION utilizes lightweight analysis to identify bug- and attack-related data, and nullifies others to protect users' privacy and reduce the report size. Our evaluation on real-world crashes shows that DESENSITIZATION can effectively remove 86.6% of all non-zero bytes in coredumps and 40.9% of those in minidumps. Additionally, the desensitized crash reports can be reduced to 45.0% of their original sizes, saving significant resources for report submission and storage.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Stephen McCamant, for their helpful feedback. We are grateful to Brendan Saltaformaggio for his valuable feedback and discussions on earlier versions of this paper. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711, ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] N. Adman and M. Satran, “Minidump Files,” https://docs.microsoft.com/en-us/windows/desktop/Debug/minidump-files, 2018.

[2] Adobe Systems Inc., “Adobe: Submitting Crash Reports,” https://helpx.adobe.com/photoshop/kb/submit-crash-reports.html, 2018.

[3] S. Andersen and V. Abella, “Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies,” 2004.

[4] S. Andrica and G. Candea, “Mitigating Anonymity Challenges in Automated Testing and Debugging Systems,” in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, San Jose, CA, 2013.

[5] Apple Inc., “Technical Note TN2123: CrashReporter,” https://developer.apple.com/library/content/technotes/tn2004/tn2123.html, 2018.

[6] H. Argp, “Pseudomonarchia jemallocum,” http://www.phrack.org/issues/68/10.html, 2012.

[7] P. Argyroudis and C. Karamitas, “Exploiting the jemalloc memory allocator: Owning Firefox’s heap,” in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2012.

[8] Backtrace, “Crash Deduplication: Triaging Effectively,” https://backtrace.io/blog/engineering/crash-deduplication/, 2017.

[9] E. Bendersky, “Parsing ELF and DWARF in Python,” https://github.com/eliben/pyelftools, 2011.

[10] blackngel, “Malloc des-maleficarum,” http://phrack.org/issues/66/10.html, 2009.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack.” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hong Kong, China, Mar. 2011.

[12] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, “Privacy-preserving Remote Diagnostics,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007.

[13] P. Broadwell, M. Harren, and N. Sastry, “Scrash: A System for Generating Secure Crash Information,” in *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[14] A. Budi, D. Lo, L. Jiang, and Lucia, “*kb*-Anonymity: A Model for Anonymized Behaviour-preserving Test and Debugging Data,” in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011.

[15] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[16] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, “EXE: A System for Automatically Generating Inputs of Death using Symbolic Execution,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006.

[17] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, “Mapping Kernel Objects to Enable Systematic Integrity Checking,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[18] N. Carlini and D. Wagner, “ROP is Still Dangerous: Breaking Modern Defenses.” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[19] M. Castro, M. Costa, and J.-P. Martin, “Better Bug Reporting with Better Privacy,” in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[20] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing Mayhem on Binary Code,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[21] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, “ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks,” in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems,” in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[23] T. O. Chris Evans, “The Poisoned NULL Byte,” https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html, 2014.

[24] J. Clause and A. Orso, “Camouflage: Automated Anonymization of Field Data,” in *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2007.

[25] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, “REPT: Reverse Debugging of Failures in Deployed Software,” in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018.

[26] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, “RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, Texas, May 2016.

[27] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, “ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, Jun. 2012.

[28] DEF CON Communications, Inc., “DEF CON Media Server,” https://media.defcon.org/, 2015.

[29] S. Designer, “Non-executable User Stack,” 2000.

[30] A. D. Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “How the ELF Ruined Christmas,” in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[31] J. N. Ferguson, “Understanding the Heap by Breaking It,” in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2007.

[32] g463, “The Use of set_head to Defeat the Wilderness,” http://phrack.org/issues/64/9.html, 2007.

[33] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, “Debugging in the (Very) Large: Ten Years of Implementation and Experience,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[34] W. Gloger, “Ptmalloc2 - a Multi-thread malloc Implementation,” https://github.com/emeryberger/Malloc-Implementations/tree/master/allocators/ptmalloc/ptmalloc2, 2001.

[35] Google, “BreakPad: A Set of Client and Server Components which Implement a Crash-reporting System,” https://chromium.googlesource.com/breakpad/breakpad/.

[36] ——, “The Chromium Projects: Reporting a Crash Bug,” https://www.chromium.org/for-testers/bug-reporting-guidelines/reporting-crash-bug, 2018.

[37] M. Graziano, D. Balzarotti, and A. Zidouemba, “ROPMEMU: A Framework for the Analysis of Complex Code-reuse Attacks,” in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi’an, China, May–Jun. 2016.

[38] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel, “Improved Error Reporting for Software That Uses Black-box Components,” in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, Jun. 2007.

[39] C. Han, “A collection of JavaScript engine CVEs with PoCs,” https://github.com/tunz/js-vuln-db.

[40] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[41] Huku, "Yet Another free() Exploitation Technique," http://phrack.org/issues/66/6.html, 2009.

[42] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards Efficient Heap Overflow Discovery," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[43] jp, "Advanced Doug lea's malloc exploits," http://phrack.org/issues/61/6.html, 2003.

[44] C. Karamitas, "Python Bindings for Intel's XED," https://github.com/huku-/pyxed, 2014.

[45] S. Kim, T. Zimmermann, and N. Nagappan, "Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, Washington, DC, 2011.

[46] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing Use-after-free with Dangling Pointers Nullification," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[47] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[48] B. Liblit and A. Aiken, "Building a Better Backtrace: Techniques for Postmortem Program Analysis," 2002, technical report.

[49] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[50] Linux, "printf(3) - Linus Man Page," https://linux.die.net/man/3/printf, 2018.

[51] Linux Programmer's Manual, "core - core dump file," http://man7.org/linux/man-pages/man5/core.5.html.

[52] P. Louro, J. Garcia, and P. Romano, "Multipathprivacy: Enhanced privacy in fault replication," in *Proceedings of the Ninth European Dependable Computing Conference*, 2012.

[53] J. Matos, J. Garcia, and P. Romano, "Reap: Reporting Errors using Alternative Paths," in *Proceedings of 2014 European Symposium on Programming Languages and Systems*, 2014.

[54] ——, "Enhancing Privacy Protection in Fault Replication Systems," in *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[55] Microsoft, "!analyze," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-analyze, 2018.

[56] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically Identifying Known Software Problems," in *IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 433–441.

[57] Mozilla, "Socorro," 2010, https://github.com/mozilla-services/socorro.

[58] ——, "Mozilla Crash Reporter," https://support.mozilla.org/en-US/kb/mozillacrashreporter, 2018.

[59] ——, "Bugzilla," https://www.bugzilla.org/, 2019.

[60] ——, "CVE-2019-9810," https://www.mozilla.org/en-US/security/advisories/mfsa2019-10/#CVE-2019-9810, 2019.

[61] J. Newsome and D. X. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2005.

[62] Offensive Security, "Exploits Database by Offensive Security," http://www.exploit-db.com/, 2018.

[63] N. L. Petroni Jr and M. Hicks, "Automated Detection of Persistent Kernel Control-flow Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007.

[64] P. Phantasmagoria, "Exploiting The Wilderness," http://seclists.org/vuln-dev/2004/Feb/25, 2004.

[65] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level Polymorphic Shellcode Detection using Emulation," in *Proceedings of the 3th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Berlin, Heidelberg, Germany, Jul. 2006.

[66] ——, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," in *Proceedings of the 10th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Berlin, Germany, Sep. 2007.

[67] M. Polychronakis and A. Keromytis, "ROP Payload Detection using Speculative Code Execution," in *Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct. 2011.

[68] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A Low-overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[69] K. Satvat and N. Saxena, "Crashing Privacy: An Autopsy of a Web Browser's Leaked Crash Reports," *arXiv preprint arXiv:1808.01718*, 2018.

[70] A. Schroter, N. Bettenburg, and R. Premraj, "Do Stack Traces Help Developers Fix Bugs?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, Big Sky, MT, May 2010.

[71] F. Schuster, T. Tendyck, C. Liebchen, L. Dvai, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications." in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[72] SecurityFocus, "SecurityFocus," http://www.securityfocus.com/, 2010.

[73] 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)*, Alexandria, VA, Oct.–Nov. 2007.

[74] Shellphish, "Educational Heap Exploitation," https://github.com/shellphish/how2heap, 2016.

[75] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[76] 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 2008 International Conference on Information Systems Security*, 2008.

[77] st4g3r, "House of Einherjar - Yet Another Heap Exploitation Technique on GLIBC," https://github.com/st4g3r/House-of-Einherjar-CB2016, 2016.

[78] Ubuntu, "Apport Crash Duplication Detection," https://blueprints.launchpad.net/ubuntu/+spec/apport-crash-duplicates, 2007.

[79] ——, "Ubuntu: CrashReporting," https://wiki.ubuntu.com/CrashReporting, 2018.

[80] R. Wang, X. Wang, and Z. Li, "Panalyst: Privacy-Aware Remote Error Analysis on Commodity Software," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul.–Aug. 2008.

[81] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[82] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[83] M. Zalewski, "AFL-Fuzz: Crash Exploration Mode," https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html, 2018.

Appendix

A. PoCs and Crash Distribution

Benchmark	Bugs	Crashes
ffmpeg (21)	#4294, #4931, #4933, #4969, #5098, #5099, #5210, #5487, #5528, #5683, #5736, #5752, #6107, #6303, #6491, #6640, #6804, #6873, #6887	1,667
	CVE-2016-10190, **CVE-2016-10191**	400
php (17)	#71311, #71436, #71449, #71450, #71637 (CVE-2016-4344), #71735 (CVE-2016-3132), #72099 (CVE-2016-4539), #72403, #72595, #73029 (CVE-2016-7417), #73240, #74577, #74593, #74604 (CVE-2017-9118), #74977	1,313
	CVE-2015-8617, **CVE-2016-4071**	200
chakra (32)	CVE-2017-11764, CVE-2017-11799, CVE-2017-11839, CVE-2017-11841, CVE-2017-11870, CVE-2017-11873, CVE-2017-11893, CVE-2017-11911, CVE-2017-8548, CVE-2017-8636, CVE-2017-8640, CVE-2017-8646, CVE-2017-8656, CVE-2017-8671, CVE-2017-8729, CVE-2017-8755, CVE-2018-0758, CVE-2018-0767, CVE-2018-0769, CVE-2018-0776, CVE-2018-0776, CVE-2018-0780, CVE-2018-0834, CVE-2018-0835, CVE-2018-0837, CVE-2018-0840, CVE-2018-0860, CVE-2018-0953, CVE-2018-8279, CVE-2018-8291	1,497
	CVE-2016-0193, **CVE-2017-0266**	100
firefox (16)	#1389812, #1397642, #1530958, #1532599, #1536768, #1538120, #1544386, CVE-2018-12386, CVE-2018-12387, CVE-2018-5091, CVE-2018-5092, CVE-2018-5095, CVE-2018-5130, CVE-2019-9791, CVE-2019-9813	30
	Alex Top 1500*	3000*
	CVE-2019-9810	50

TABLE VI: The bugs for generating crashes of various real-world benchmarks, along with the number of crashes collected. The rows in bold indicate bugs used for generating attacker-driven crashes. The row with asterisk superscript for firefox indicates the crashes caused by SIGABRT signals when visting the websites. Visit `https://trac.ffmpeg.org/ticket/[BUGID]` and `https://bugs.php.net/bug.php?id=[BUGID]` for the detailed reports of the `ffmpeg` and `php` bugs, respectively. The PoCs of all the `chakra` and `firefox` bugs can be found at [39] and [59].

B. Cases of Privacy Leakage

```
struct nsCookieAttributes {
  nsAutoCString name, value, host, path; ...
}
nsresult CookieServiceChild::SetCookieStringInternal(...){
  // loop to parse cookie string data
  do {
    nsCookieAttributes cookieAttributes;
    moreCookies = nsCookieService::CanSetCookie(
        aHostURI, key, cookieAttributes, requireHostMatch,
        cookieStatus, cookieString, serverTime, ...)
    // more actions...
  } while (moreCookies);
}
```

Fig. 13: The code of handling cookie strings in `firefox`. The shaded line may leave user's cookies in crash reports.

```
nsresult FSMultipartFormData::AddNameValuePair(
    const nsAString& aName, const nsAString& aValue) {
  nsAutoCString encodedVal;
  nsresult rv = EncodeVal(aValue, encodedVal, false);
  // prepare more headers & data...
}
nsresult HTMLFormSubmission::GetFromForm(HTMLFormElement* aForm,
    HTMLFormSubmission** aFormSubmission, ...) {
  // Get encoding
  auto encoding = GetSubmitEncoding(aForm)->OutputEncoding();
  // Choose encoder
  if (method == NS_FORM_METHOD_POST &&
      enctype == NS_FORM_ENCTYPE_MULTIPART) {
    // create headers & data for FSMultipartFormData;
    // in turn calls AddNameValuePair()
    *aFormSubmission = new FSMultipartFormData(actionURL,
        target, encoding, aOriginatingElement);
  // Else, other encoders ...
}
```

Fig. 14: The sample code of preparing form data in `firefox`.

```
nsresult nsAutoCompleteController::EnterMatch(
    bool aIsPopupSelection, dom::Event* aEvent){
  nsCOMPtr<nsIAutoCompleteInput> input(mInput);
  nsCOMPtr<nsIAutoCompletePopup> popup(GetPopup());

  int32_t selectedIndex;
  popup->GetSelectedIndex(&selectedIndex);
  bool shouldComplete;
  input->GetCompleteDefaultIndex(&shouldComplete);

  nsAutoString value;
  if (selectedIndex >= 0) {
    // selected index for popup ...
    GetResultValueAt(selectedIndex, true, value);
  } else if (shouldComplete) {
    // incomplete input triggering matched entry
    nsAutoString defaultIndexValue;
    if (GetFinalDefaultCompleteValue(defaultIndexValue))
      value = defaultIndexValue;
  // else...
}
```

Fig. 15: The sample code of autofilling per history inputs in `firefox`.