Towards Automated Discovery of Crash-Resistant Primitives in Binary Executables

Benjamin Kollenda[1] Enes Göktaş[2] Tim Blazytko[1] Philipp Koppe[1] Robert Gawlik[1] R.K. Konoth[2]
Cristiano Giuffrida[2] Herbert Bos[2] Thorsten Holz[1]

[1] Horst Görtz Institut for IT-Security (HGI), Ruhr-Universität Bochum, Germany
[2] Computer Science Institute, Vrije Universiteit Amsterdam, The Netherlands

***Abstract*—Many modern defenses rely on address space layout randomization (ASLR) to efficiently hide security-sensitive metadata in the address space. Absent implementation flaws, an attacker can only bypass such defenses by repeatedly probing the address space for mapped (security-sensitive) regions, incurring a noisy application crash on any wrong guess. Recent work shows that modern applications contain idioms that allow the construction of *crash-resistant code primitives*, allowing an attacker to efficiently probe the address space without causing any visible crash.**

In this paper, we classify different crash-resistant primitives and show that this problem is much more prominent than previously assumed. More specifically, we show that rather than relying on labor-intensive source code inspection to find a few "hidden" application-specific primitives, an attacker can find such primitives *semi-automatically*, on *many classes of real-world programs*, at the *binary level*. To support our claims, we develop methods to locate such primitives in real-world binaries. We successfully identified 29 new potential primitives and constructed proof-of-concept exploits for four of them.

I. Introduction

While arguably a weak defense by itself [42], address space layout randomization (ASLR) plays a pivotal role in almost all modern defenses that hide sensitive information at a random location in memory. ASLR can be categorized as a basic form of information hiding, namely randomizing the location of code images, heaps and stacks in the address space. However advanced defenses increasingly rely on the information hiding primitives provided by ASLR to (pseudo-)protect sensitive data such as encryption keys [31], code pointers [17], [28], and redirection tables [9]. If implemented properly, even attackers with full read-write access over the process' memory will not be able to access the sensitive data, because they are tucked away at random memory locations in a huge address space. Since the process memory will not contain a single pointer to the hidden region(s), the only way for an adversary to get to the secrets is by trial-and-error. It is almost certain that such attempts will quickly access unmapped memory, which normally incurs a crash. Hence, the sensitive information is believed to be safe from attackers [31].

State-of-the-art attacks on information hiding try to reduce the entropy of the randomization as much as possible. For instance, they trick the program into increasing the size of the hidden region [24], or into performing gigantic memory allocations [35], or into leaking information via sophisticated timing side channels [21]. However, unless the attackers reduce the entropy to zero, the final step in these attacks on randomization still relies on trial and error, with a high likelihood of crashes.

Today's successful attacks against the residual entropy build on the observation that various server applications automatically restart upon a crash, enabling an attacker to repeatedly probe the address space in a brute-force manner [13]. Hence, recent work proposed a variety of improvements to mitigate the attacks [30]. In this paper, we assume that information hiding is *perfect*, all of the proposed improvements against disclosure attacks are in place, the attacker cannot completely drop all entropy, and the only way to find sensitive information is by performing a crashless brute-force attack. As a result, an attacker needs to find novel crashless ways to bypass such sophisticated defenses.

If the target application has code fragments that do *not* crash when reading from or writing to inaccessible memory (because they handle such violations themselves, say, in an exception handler), attackers may use these fragments to probe for the secret information repeatedly. This technique was introduced by Gawlik et al. [22]. We define *crash-resistant code* as code that will not crash the program upon an invalid memory access. It is markedly different from *crash-tolerant code* where a server application immediately re-forks worker processes or a web browser re-opens a tab *after a crash* happened. While crash tolerance can serve as a vector for attacks, it is much noisier and thus less attractive than crash resistance—thousands of crashes in a short amount of time may easily raise alarms in real-world scenarios. In contrast, crash resistance incurs no crashes at all and is therefore much stealthier.

Note that crash resistance in its intended form is a classic double-edged sword. On the one hand, it enhances software reliability and enables applications to automatically recover from malformed inputs that cause an access violation, also improving user experience. On the other hand, it permits attackers to abuse the crash-resistant code snippets, dubbed *memory oracles*, to probe the address space [22] and even entropy-reducing attacks have used such crash-resistant probes [24]. Note that the underlying principles of each memory oracle can vary greatly. They can range from system level exception handlers, over system calls to application specific exception handling.

Unfortunately for attackers, finding crash-resistant primitives

in real-world software is labor-intensive, manual work. Given a new application, finding such a specific primitive is difficult and rare, especially in the absence of source code. Thus, in its original form, the approach was difficult to use in a generic way across many applications. Furthermore, the concept was thought to be only applicable to client applications.

In this paper, we present semi-automated methods to locate crash-resistant primitives in a given binary executable and generalize the basic concept of crash resistance by demonstrating that the method is also applicable to server applications. Based on our observation of the root cause for the crash-resistant primitives, we developed two different strategies on how to locate further instances of them in binary executables. Both serve as a starting point for fully automated identification of such primitives and we demonstrate that our methods can find them quickly in a number of real-world server and client applications on different platforms. While our techniques do not produce fully-fledged memory oracles in an automated way, we substantially reduce the engineering work required to analyze a given binary executable. More specifically, we are able to identify in an automated way code constructs that can serve as crash-resistant primitives.

The first approach targets the interface between a user-mode program and the kernel, such as system calls on Linux or API calls on Windows. This builds on the intuition that many such calls allow the kernel to respond to an invalid user address given as a parameter by returning an error code (without crashing) to user space. We leverage taint analysis to track which bytes in attacker-controlled memory eventually determine the appropriate parameters in calls that fit crash resistance (e.g., system calls that return `-EFAULT` on access faults). Intuitively, by modifying these memory locations, the attacker may probe the address space—assuming that the program does not also dereference the address outside the crash-resistant code fragment. We explore this idea for both server applications on Linux and client applications on Windows to study if this approach is feasible in practice.

The second approach targets exception handling code, since this is a common technique to guard program code and respond to error conditions—hence a prime candidate for crash-resistant code. We look for code structures that paper over access violations, thus yielding candidates for crash-resistant code which we subsequently vet. In a first step, we extract the exception handlers from a binary and then use symbolic execution to determine which ones handle access violations. Given that exception handling is commonly used for client applications on Windows [22], we focus our analysis on such programs.

Using these two methods we successfully found 29 new crash-resistant primitives in popular server applications and web browsers. We also developed four primitives found in Nginx 1.9, Lighttpd 1.4, Internet Explorer 11 and Firefox 46 into proof-of-concept exploits to demonstrate the effectiveness of our approach.

In summary, we make the following four contributions:

- We classify known crash-resistant primitives based on their underlying mechanisms and use these properties as a way to identify additional instances of memory oracles.
- We show that it is possible to discover (otherwise extremely hard-to-find) crash-resistant code primitives in an automated fashion in both client applications on Windows and servers on Linux.
- We are the first to find and use crash-resistant code primitives on server applications. In contrast to crash-tolerant approaches, which simply exploit the fact that server applications typically restart upon a crash, our technique offers much more flexibility and stealthiness for an adversary.
- We evaluated our techniques on five popular servers and two browser applications and found 29 new crash-resistant primitives, of which we developed four into fully fledged proof-of-concepts. In addition, we discuss how attackers can exploit these primitives to bypass any defense utilizing information hiding.

II. Background and Related Work

In the following, we provide a brief overview of the technical concepts we use in the rest of this paper to classify crash-resistant primitives and detect them in an automated way.

A. Crash-resistant Primitives

Several papers [13], [21], [41] have shown that server applications are vulnerable to guessing attacks against defenses based on randomization due to their crash-tolerant nature, i.e., a network service typically restarts in an automated way upon a crash. This enables an attacker to perform brute-force attacks and eventually reach her goal. On the downside, the induced crashes are noisy and a defender might easily notice a server application crashing thousands of times in a small amount of time. Such attacks relying on crash-tolerant code were believed to not affect client applications given their hard crash policy: client programs usually do not restart after a crash and thus an attacker is limited to a single try to bypass a given defense.

A new twist on crashes was proposed by Gawlik et al. [22], who demonstrated that so called *memory oracles* can be leveraged to probe arbitrary memory regions and discover reference-less memory. The authors showed two examples of such primitives: one usable in Internet Explorer which abuses a system feature, and one in the 64bit version of Firefox which is based on a program specific performance optimization involving exception handlers. Furthermore the authors noted that some system calls, like `access`, might be usable as memory oracles. In this paper we build on these findings to define categories of memory oracles and develop tools that aid in the search for similar primitives.

B. Information Hiding Defenses

Software-based fault isolation (SFI [20], [45]) is a technique that allows code to be executed with strong safety and security guarantees by adding checks to critical operations such as memory accesses or control flow transfers. Similarly, techniques like SoftBound [34] or baggy bounds checking [7]

enable memory safety, preventing many attack vectors in a generic way. Unfortunately, the overhead induced by such approaches is prohibitively high in practice [43], which prevents a widespread adoption. As a more efficient alternative, several recent defenses [9], [17], [28], [31] rely on information hiding to prohibit an attacker from obtaining valuable information such as encryption keys, code pointers, and redirection tables.

However, any defense based on information hiding is at risk in the presence of crash resistance. Most prominently, address space layout randomization (ASLR) can be bypassed as it is possible to either locate the memory location of a library directly or instead locate otherwise reference-less structures that contain pointers to loaded binary images. For example, on Windows this is the case for thread information blocks (TEBs) and process environments blocks (PEBs), two data structures which allow an attacker to retrieve the location of all loaded modules. In practice, however, information leaks providing the location of code images are common. In contrast, inferring addresses of data structures belonging to stronger defenses beyond ASLR is not commonly possible. Such advanced defenses typically assume an attacker equipped with an information leak—and thus full knowledge of the memory layout of the process—with the exception of the meta data structures of the defense in question. Commonly, this is implemented by only keeping the addresses to these structures in a register and preventing any write of this value to memory. Without knowledge of the exact location, an attacker can thus not overwrite the data and the defense can enforce certain properties on the protected program. Examples of such advanced defenses are Code-Pointer Integrity (CPI) [28] and any Control Flow Integrity (CFI) solution relying on a shadow stack [14], [17] to protect backwards edges. Armed with crash resistance, an attacker can locate the hidden region of CPI (both the sparse region and the hash-table based implementation [22]) and modify the metadata of any pointer. This means that the main assumption of CPI, namely that the pointer metadata of any code pointer cannot be modified by an attacker and thus any use of crafted pointers is prevented, no longer holds true. A similar attack is possible for shadow stack-based CFI solutions. These solutions hide the location of the shadow stack from the attacker by using, for example, a dedicated register or thread local storage. If an attacker can find the stack via crash-resistant probing, she can modify the information stored there.

The same holds true for implementations that instead of using a dedicated shadow stack rely on separating safe and unsafe stacks. The SafeStack [28] implementation by CPI, which is now included in LLVM, provides such a feature and uses the native stack only for statically proven safe variables. This means an attacker cannot overflow a local stack variable to overwrite return addresses. Additionally, the compiler ensures that no references to the SafeStack are written to memory outside of the SafeStack itself, resulting in a reference-less memory region. As return addresses are located on the SafeStack, a control-flow hijack based on overwriting return addresses is no longer applicable, without being able to pivot the stack pointer. However, as crash resistance allows scanning the whole address space, the stack can be located [24] and the return addresses overwritten.

Another recent defense that aims to protect against meaningful control flow hijacks is ASLR-Guard [31]. The main concept is that an attacker is rendered unable to retrieve a plain text code pointer, so any control flow hijack attack is reduced to pure chance. This is achieved by (i) removing links between data pointers and code pointers and (ii) encoding any code pointers stored to data memory. The first countermeasure ensures that a data pointer leak (which is explicitly allowed within the threat model) gives no indication of the location of executable code. With the common form of ASLR the address of the data section, which can be located by the attacker, allows inferring the location of executable code of the same module by simply adding a static offset. The first countermeasure is combined with a pointer protection scheme that never writes plain text code pointers to data locations. This includes most prominently the stack, which can be located and leaked by an adversary. However, using a crash-resistant primitive, it is possible to just probe memory until the executable code is found. An attacker does not need to infer the location from either data pointers or saved code pointers: after the executable code has been located, known attacks like JIT-ROP [42] can be used again.

Apart from models that merely restrict an attacker's knowledge of the memory layout, some defenses impose additional properties on the memory. The principle of execute-only memory (XoM) [8], [23] allows for an additional access right, in contrast to the standard of execute access implying read access. Without the possibility of reading the code, an adversary has no way of determining the actual bytes used to implement a given functionality. If this technique is combined with some form of fine-grained ASLR, it prevents code-reuse attacks requiring knowledge about the exact code either statically, e.g. ROP [38], or at run time, e.g. JIT-ROP. Against these defenses, memory oracles are less useful as probing primitives are mainly utilized to locate memory, whereas XoM does not use any information hiding in this regard. As such, when using plain probing attempts, the result of trying to read code would still indicate inaccessible or unmapped memory. However, with crash-resistant primitives that allow broad capture of any exception, it could be possible to brute-force the code layout. A similar attack might be possible against sophisticated defense solutions such as Readactor [15] and Readactor++ [16], which focus on both hiding code pointers from an attacker and enforcing XoM.

An important type of defense that can hamper the success of memory probing is runtime re-randomization [9], [12]. Employing runtime re-randomization can substantially decrease the success probability of either the scanning itself or the following attack step. Due to the "moving target", it is harder for an attacker to locate the code she needs and at the same time abuse it within the time constraints given by the defense. However, crash-resistant primitives that allow invalid executions to be recovered can also weaken the security guarantees of

these defenses: given enough tries, such schemes can likely be bypassed due to the chance of using the right randomization in the attack attempt.

C. Threat Model

In the following section, we introduce several techniques to detect crash-resistant code within a given binary executable. Several conditions must hold for these code snippets to be useful for an attacker. To this end, we assume the following threat model in the rest of this paper, which is realistic and matches the capabilities of a real-world attacker. Further, it is consistent with recent research [15], [16], [18], [33], [40], [42]:

- **Arbitrary read/write primitive**: The attacker can read from and write to arbitrary memory locations.
- **Information leak**: An information leak allows the adversary to infer the location of data protected via some kind of randomization scheme such as ASLR. For example, the adversary can locate the base address of module locations, but she cannot access reference-less memory locations.
- **Computational capabilities**: The attacker can perform computations during the attack. This can be some form of scripting environment on the client side or a server accepting multiple connections. The latter allows an attacker to query the state of the server with one connection and act on this information with another one.
- **Writable $\oplus$ Executable memory**: Memory pages are marked as either executable or writable, but not both at the same time.
- **State-of-the-art defenses**: The target application employs some kind of state-of-the-art defense to thwart code-reuse attacks. This can be either an information hiding scheme such as a shadow stack-based CFI approach, or some kind of defense to prevent control-flow hijacking attacks such as CPI.
- **Hard crash policy**: The application does not automatically restart after a crash. This includes automatically restarting a crashed worker process or a user opening a website again.

III. High-Level Overview

In code-reuse attacks, the exploitation procedure of memory corruption vulnerabilities can be subdivided into three phases. Initially, the attacker leverages a memory corruption vulnerability to establish a read/write primitive. Depending on the kind of vulnerability, an attacker may be able to read some out-of-bounds bytes in order to disclose some information about the address space layout, or leverage some other kind of information leak. In the second phase, the attacker prepares the payload, for instance, by relocating a static ROP chain [38], the counterfeit objects of the COOP attack [40], or by compiling a JIT-ROP chain [42]. Meanwhile, it may be necessary for the attacker to also bypass code-reuse defenses such as fine-grained randomization [11], [26], [46], shadow stack-based CFI solutions [14], [17], CPI [28], and other information hiding-based approaches [9], [18], [31]. Finally, the attacker hijacks the control flow by overwriting code pointers or other sensitive pointers. In practice, it may be necessary to carry out (parts of) the sequence multiple times in order to bypass multi-process sandboxing schemes or to escalate the privileges of the user.

We assume that a defense relying on information hiding, for example one of those discussed in Section II-B, is employed by the target application. Thus an attacker needs to leverage a crash-resistant primitive in the second attack phase with the help of the following steps (Figure 1):

1) *Overwrite a value in memory*: the attacker uses a memory corruption primitive to prepare the memory for the next step, usually overwriting pointers to data which are then probed later; modifying data can cause usually benign functions to exhibit unintended/malicious behavior.
2) *Trigger execution of probing*: the attacker forces the program to execute the probing primitive. This is trivial via a control-flow hijacking attack, but we focus on locating primitives legitimately accessible to the attacker, e.g., functions in a scripting environment.
3) *Infer the state of the probed location*: finally, the attacker requires an indication whether or not the probing attempt succeeded. In the easiest case, this is directly inferred from a return value or similar information, but usually the attacker needs to infer the state indirectly, e.g., via memory changes or execution timings.

These steps can be repeated several times to probe other memory locations until enough information about the memory layout is known to the attacker.

In the remainder of this section, we describe our classification of crash-resistant primitives and outline how we locate additional candidates in a (semi-)automated way for each type. Note that we do not cover functionality intended for querying the memory layout of a process, such as the `/proc` file system under Linux or functions like `VirtualQuery` or `IsBadReadPtr` on Windows.

A. Syscalls and OS API Functions

Modern operating systems allow for the quasi-parallel execution of different, isolated user space processes. This also means that a fault in a single program must not cause another independent program or the whole system to fail. This is achieved by handling errors, e.g., invalid memory accesses, on a per-program basis. However, once a program needs to pass data to the operating system, any error in this data (or often in the case of a memory error, in the location of this data) can

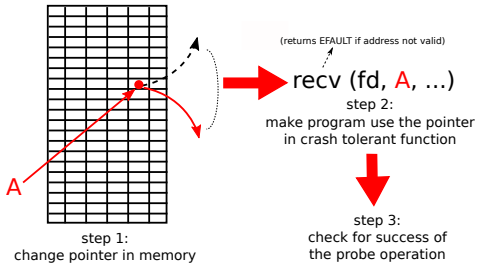

Figure 1: Attacker's procedure to probe memory without crashing

potentially impede the stability of the whole system. As such, whenever data is passed from user space to kernel space, the OS must perform strict error checking.

To allow an application to react to an error in its data, a failure state is usually returned. If this event is properly handled, the application can resume execution. However, in the case of a scanning attempt, this property can leak valuable information to the attacker. If she is able to influence the data in a way that causes a failure state to be reported for invalid addresses and success for valid ones, she is able to probe the address space and uncover hidden memory regions. The actual implementation of such a primitive is heavily OS dependent, however it is applicable to any program allowing manipulation of input data and inferring the error state afterwards.

On Windows systems, the OS exposes a set of system API functions which are then translated to system calls after preprocessing in user space. The result is an often heavily abstracted interface to the underlying syscalls. This is problematic in the context of memory probing, because any access to the supplied memory region can lead to a fault in user space, preventing the OS from gracefully reporting the error state to the program. In contrast to this, programs running on Linux are free to access syscalls directly or with minimal abstraction. While both system operate on a similar principle—a specified interface is provided for the user programs—we had to account for the differences and chose to develop different techniques for the Windows system API and Linux syscalls.

1) Linux syscall interface: The Linux kernel exposes a set of well documented syscalls to the user space. These are used to perform kernel-level functionalities from user space, such as file- or network-related operations, and memory management. In case an error occurs during a syscall, the kernel returns `-1` to the user process to indicate that something went wrong and assigns the appropriate error code to the `errno` variable in user space.

Several syscalls require the application to provide pointers to memory in user space such that the kernel can read data from or write data to that area. In case the address is invalid, the kernel sets the `errno` variable to `EFAULT` [5], which indicates that the memory location is not accessible. `EFAULT` is a common error code that many popular system calls use. Examples include `connect`, `read`, `write`, `epoll_wait`, `recvfrom`, `open`, and many others. If an attacker has control over the memory address of the relevant syscall parameter, she can potentially probe the address space for accessible memory areas without crashing the application.

For instance, many servers contain a main loop like this:

```
1  while (true) { // server loop
2    ...
3    if (read (fd, buf, MAX_BUF_LEN) < 0) {
4      terminate_connection(fd, "read failed");
5      continue;
6    }
7    ...
8  }
```

Listing 1: Server loop with error handling

An attacker who is able to control the `buf` pointer can provide any address and discover whether or not it is valid. Note that the server will not crash.

The detection of crash-resistant candidates can be automated as follows. Because of their relevance for crash-resistant probing candidates, we maintain a list of all syscalls that may return `EFAULT` and monitor their occurrence during an instrumented, automated execution. We use taint analysis to identify which parameters can be influenced by an attacker and execute unit tests to analyze which syscalls can potentially be executed during a run of the application. As a result, we obtain a list of potential candidates.

2) Windows API: In contrast to Linux, user-mode applications on Windows exclusively utilize the API provided by the operating system [39]. As a result, Windows never exposes the system calls directly to the application. Nevertheless, the same method outlined above can be applied: if an API function accepts a pointer as an argument and an attacker can control this pointer, she can point it to arbitrary memory addresses and observe the return value or side effects to infer the resulting state. An example for this concept is the API function `VirtualQuery`, it is used to obtain information on the state of a memory address. If an attacker is able to control the argument `ptr` and knows the location of `mem_info`, she can probe any page for its state and permissions:

```
1  void* ptr = NULL;
2  PMEMORY_BASIC_INFORMATION mem_info = malloc(sizeof(
       MEMORY_BASIC_INFORMATION));
3  ...
4  VirtualQuery(ptr, mem_info, sizeof(
       MEMORY_BASIC_INFORMATION));
5  ...
```

Listing 2: Example for VirtualQuery API call on Windows

While `VirtualQuery` is trivially able to serve as a memory oracle, the functions targeted by our framework do not explicitly state their crash-resistant nature in the documentation. Therefore we need to locate them ourselves using the following steps. First, we reduce the set of all available Windows API functions to only those functions with crash-resistant properties. For this we apply a basic form of fuzzing to the Windows API functions. In the second step, we attempt to find code paths to these crash-resistant API functions by harvesting API calls, tracing instructions, and filtering the results via custom analysis scripts. Third, we classify the pointer arguments of the crash-resistant API functions to figure out if we can actually control the pointer on the execution path: only if an attacker-controllable pointer is found, we can construct a corresponding crash-resistant primitive.

B. Exception Handlers

Another feature of common operating systems and programming languages is allowing a program to recover from an exception. These exceptions can range from a software generated exception to hardware faults. For our purposes, the possibility of handling an invalid memory access and resuming execution afterwards is especially critical. Low-level languages like C/C++ allow a programmer to explicitly add constructs

to catch faults and tell the operating system how to resolve them. Essentially the operating system or language runtime provides information about the fault to a specific subroutine in the program which then can choose from a set of options.

Commonly these options include simply ignoring the exception, executing the next instruction as if nothing happened, resuming execution at a different location instead, or passing the exception along to another handler. In the latter case, if the exception is not handled by the program, the OS will usually terminate the program. A crash-resistant primitive using exception handling requires the program to dereference an attacker controlled pointer inside a code block that is covered by an exception handler. After a probing attempt, the result must be visible to the attacker, either explicitly by a return value or similar values, or implicitly with the help of side channels (e.g., timing). In addition, the exception handler must allow memory faults to be handled, which can be excluded using the information provided for each fault to filter out unsupported exceptions.

Under Linux, exception handling is implemented as signal handling. A signal is a software interrupt that can be handled by a process in three ways: (i) it can be ignored, (ii) it can be caught by a signal handler, or (iii) the signal's default action can be performed [6]. For example, the default action of the signal `SIGSEGV` (i.e., segmentation fault or access violation) is the termination of the process.

In contrast, Windows utilizes two techniques for exception handling, *Structured Exception Handling* (SEH) [36], [39] and its extension *Vectored Exception Handling* (VEH) [37]. While SEH-based exception handlers operate locally on a guarded function, VEH-based exception handlers can be used globally within the process. The following example shows a SEH-guarded block with its corresponding *filter*:

```
1   __try {
2       ...
3       // guarded code
4       value = *ptr;
5       ...
6   }
7   __except(EXCEPTION_EXECUTE_HANDLER)
8   {
9       value = -1;
10  }
```

Listing 3: Example for Structured Exception Handling (SEH) with corresponding filter on Windows

This exception handler could be used to probe memory addresses, if an attacker controls the value of `ptr`. In the case of any exception, including access violations, `value` would be set to `-1`; if the location is readable, it would be set to the content of that address. Specifying `EXCEPTION_EXECUTE_HANDLER` as the filter expression allows the handler to be executed for all types of exceptions.

Any exception leads to the evaluation of the corresponding *filter expression* which determines the appropriate action. The filter can (i) simply resume the program execution (i.e., ignore the exception completely), (ii) transfer the control flow to the exception handler, or (iii) forward the exception to the next handler. The filters are implemented as separate functions on the binary level and referenced in SEH structures. If an exception is not handled and it is considered fatal by the operating system, it causes program termination. This is also the case for access violations generated by scanning attempts. If an attacker can control dereferences inside of a guarded code block and the corresponding filter allows either handling or ignoring of access violations, she is able to scan arbitrary memory.

To automatically locate potential memory oracles, we first need to collect all available exception handlers and their guarded code regions. This is done via static analysis of the target binary. We then discard any exception handlers which are not able to handle access violations as indicated by their filters. For this we symbolically execute the filter functions. The resulting set of potential code locations is then analyzed with the tools described in the previous section: instead of system APIs, we now target the guarded code locations.

C. Swallowed exceptions

On further investigation we also added a third class of possible memory oracles which was not covered before. There are circumstances where exceptions are silently ignored. Here we do not consider cases where an exception handler of a program simply ignores the fault and continues execution or a system API detects an error and the user program does not check the error status. While leading to the same result, suppressing of memory faults, these methods deliver the error status to the user program, it just does not act on them. Instead swallowed exceptions give no feedback to the user program that an exception occurred. An example are user-kernel-user callbacks [10], where the exception handling mechanism can not support the context switches. The result is that the calling program has no way of detecting that an exception occurred. We do not consider this class of crash-resistant primitives in our analysis.

IV. Implementation

In the following, we briefly outline the implementation of our framework and describe the reasons for our design choices. Implementation details are available in the corresponding technical report [27].

A. Syscalls on Linux

On Linux, we use dynamic taint tracking to isolate viable candidates. We target common server applications with test cases, allowing for sufficient code coverage. As we reuse test cases and want to support additional applications with minor changes, we chose a minimally invasive approach. The server program is instrumented with libdft [25]—a data flow tracking library which we extended with byte granular taint tracking—and the corresponding client program is controlled by our monitor application. The monitor can send client and server custom commands to control the taint state and invalidate pointer arguments. After a test run we obtain a list of potential crash-resistant primitives with details on which arguments were

valid or invalid during the test run. This list is then verified manually to eliminate false positives.

B. Windows API Functions

As the Windows API does not define error states as uniformly as the Linux syscall interface, we need to isolate appropriate target functions ourselves. For this we use a fuzzing approach to gather a list of functions that handle invalid pointer arguments gracefully. Afterwards we locate any usage of these functions in the targeted applications. For this we use the dynamic instrumentation framework DynamoRIO [19]. The resulting list of call sites is then reduced using heuristics to only retain promising candidates. At the end of this analysis phase, the results need to be manually verified to exclude any false positives, mainly those cases in which the pointer arguments are either short lived stack variables or volatile heap locations, and thus cannot be controlled by the attacker.

C. Exception Handlers

Aside from the system level candidates, we also target application-specific memory oracles in the form of exception handlers. For this we use the fact that under 64bit Windows every function in an application needs to provide stack unrolling information in case it is contained in the call stack of an exception. As such, we can parse the corresponding `.pdata` section and retrieve the list of all exception handlers in an executable module. We then use symbolic execution and the SMT solver Z3 [32] to filter out the exception filters that allow either all exceptions or at least access violations to be handled. After this analysis step, we reuse the analysis methods we developed for the tracking of API functions and target the code covered in the exception handlers. At the end, we again manually verify the results.

V. Evaluation Results

Based on our prototype implementation, we now discuss the results of our analysis on binary executables for both Linux and Windows.

A. Syscalls on Linux

We evaluated our framework with widely used server applications on Linux. In particular, using our framework, we ran the standard test suites for the following server programs: Nginx 1.9, Cherokee 1.2, Lighttpd 1.4, and Memcached 1.4. We focus on such popular server programs since they all handle multiple connections per process. An attacker can simply use one connection to probe a memory address (using a discovered crash-resistant primitive) and another connection to exercise her arbitrary read/write primitives and modify the state of the probing connection.

For completeness, we also consider server programs that handle every new connection in an independent worker process, focusing our analysis on PostgreSQL 9.0. In such cases, the attacker can only use a single connection to probe and modify the state of the program. While exploitation is generally more complicated (it might be harder to restore the preferred connection state to probe a new memory address), we still found usable primitives in practice. Note that our goal is simply for the worker process not to crash and a graceful process termination is sufficient for our purposes. As the worker process is expected to terminate after serving a request, this does not constitute any abnormal action.

Table I: Syscalls indicated as potential (±) or valid (+) cr primitives by our framework on Linux. Green circled ones were manually verified to be usable as a cr primitive. The red non-circled plus sign indicates a result manually verified to be a false positive.

Syscalls	Nginx	Cherokee	lighttpd	Memcached	PostgreSQL
chmod					±
connect	±				
epoll_wait		⊕	±	+	⊕
mkdir			±		±
open		±			
read		±	⊕	⊕	±
recv	⊕	±			
recvfrom				±	
send		±			
sendmsg				±	
statfs		±			
symlink					±
unlink			±		±
write		±	±		±

Table I shows all the candidates reported by the framework. As depicted by the non-circled plus-minus sign, many of the candidates end up in a segmentation fault if we automatically alter the target memory locations with an invalid memory address. We have confirmed our framework has flagged all these cases correctly using manual inspection. Despite the many invalid candidates, our framework discovered a usable crash-resistant primitive, depicted with a green circled plus sign, in all of our server programs (with the two *potential* candidates confirmed via manual verification).

Four candidates in total were indicated as *valid* candidates by the framework. We confirmed that our framework has flagged all these cases correctly using manual inspection, except the valid candidate on Memcached which turned out to be a false positive (depicted as a red non-circled plus sign). Manual inspection revealed that the connection handling thread exits after the candidate syscall `epoll_wait` returns an error code, while the server keeps running—which our framework currently interprets as correct behavior. Subsequent connections, however, never get processed by the now terminated connection handling thread and the primitive is effectively unusable for multiple probing attempts. This false positive can be simply eliminated by checking the status of connection handling threads, a strategy which our current prototype does not yet support in a generic way. From our analysis, we found usable candidates in `recv` in Nginx, `epoll_wait` in Cherokee and PostgreSQL, and `read` in Lighttpd and Memcached. We exemplify how such candidates can be used as crash-resistant primitives in Section VI.

B. Windows API Functions

We had to first collect crash-resistant API functions on Windows. We extracted 20,672 API functions from the MSDN library, of which 11,521 (55.7%) contained at least one pointer argument. Hence, only these functions served as inputs for the analysis phase via our custom API fuzzer. As a result, we found 400 API functions that are candidates for a crash-resistance primitive both under Windows 7 and 10.

In the next step, we attempted to locate these functions on execution paths, outlined here exemplary for Internet Explorer 11 (64bit) on Windows 10. For this, we logged all calls to target API functions while visiting the top 500 websites from *alexa.com* [1]. In addition, we ran browser and JavaScript benchmarks [2], [44] to increase the code coverage. Only 25 crash-resistant API functions were found on an execution path. Finally, we used our analysis scripts to determine if these functions were triggered from a JavaScript context. We found 12 functions with this characteristic.

To be a usable crash-resistant primitive, we have to trigger them from JavaScript and control their arguments; in addition, we must be able to intercept the return value. To analyze these two properties, we created instruction traces and analyzed the resulting execution paths. Unfortunately, all candidates had to be excluded after a manual analysis since *all* of the pointer arguments were unusable for our purposes. The reasons for this are threefold. First, most functions were query functions (e.g., `GetPwrCapabilities`) which are usually called by supplying a stack-allocated structure. If such a pointer is invalid, then the stack pointer is corrupted. This leads to an illegal memory access and causes the program's termination. Second, a majority of the remaining candidates' pointer arguments were dereferenced outside of the target function. This also leads to an illegal memory access if the pointer is invalid. Third, we cannot control the pointer arguments of some candidates since the pointers were volatile heap pointers which had no previous references stored in memory.

This negative result for Windows API functions does not imply that no crash-resistant primitive can be constructed using our method. The coverage of test cases influences the number of excluded functions after code path analysis: only 25 of the 400 candidate API functions were observed on execution paths. Further work on improving code coverage may lead to more candidates, and hence yield crash-resistant primitives.

C. Exception Handlers

To test the feasibility of our approach leveraging exception handlers, we collected the executed code blocks during normal usage. Again we use Internet Explorer 11 (64bit) on Windows 10 as an example. We instrumented the browser with DynamoRIO and browsed again the top 500 websites from *alexa.com* [1]. Then, we analyzed all DLLs that have been loaded by the browser and extracted the exception handlers. Afterwards, we reduced this set by symbolically executing the corresponding filters and cross-referencing the remaining exception handlers with those that have been visited.

Table II: The number of unique code locations that are guarded by C-specific handlers during an Internet Explorer 11 run. The code locations that appear on the execution path are from the set after symbolic execution (SB).

DLL	# guarded program code		
	before SB	after SB	execution path
user32	70	63	40
kernel32	76	66	14
mshtml	129	10	3
ieframe	34	22	6
kernelbase	96	81	0
ntdll	113	65	19
jscript9	22	6	4
rpcrt4	62	20	6
sechost	133	11	0
ws2_32	82	29	10
xmllite	10	2	1

Table III: Unique exception filters in different DLLs before and after symbolic execution (SB).

DLL	# filter functions			
	before SB		after SB	
	x64	x32	x64	x32
user32	9	17	2	15
kernel32	60	7	50	3
mshtml	128	33	9	2
ieframe	29	6	17	0
kernelbase	54	21	39	19
ntdll	71	25	23	15
jscript9	19	5	3	4
rpcrt4	50	11	8	1
sechost	126	26	4	1
ws2_32	55	25	3	17
xmllite	10	0	2	0

Table II provides an overview of the amount of program code that is guarded with C-specific exception handlers for a subset of the loaded DLLs. In addition, the table shows the code locations that are guarded with crash-resistant candidates (including the exception handlers that use catch-all filters) as well as their number of occurrences on the execution path. For instance, there are 63 crash-resistant candidates from 70 exception handlers in `user32.dll`, whereby 40 code locations that are guarded by those are executed while browsing the most popular websites. Contrary, `sechost.dll` guards 133 code locations, whereby 11 crash-resistant candidates exist and no guarded code location was triggered during our test. In addition, Table III shows that symbolic execution significantly reduces the set of exception filters, since it drops the majority of filter functions given that they are not fit for our purposes. As described before, we use symbolic execution to exclude all filters that do not allow access violations to be handled. For example, only 4 of 126 filter functions remain in `sechost.dll`, while 9 of 129 are left in `mshtml.dll`. In total, we found 6,745 C-specific exception handlers in 187 analyzed DLLs. These exception filters use 5,751 different filter functions. After the symbolic execution, 808 filters remain that handle access violations, including catch-all filters. These filter functions are

used by 1,797 exception handlers.

In the next step, we cross-referenced the visited code blocks with those filtered exceptions. These exception handlers may lead to crash-resistant primitives that are known to be triggered. In absolute numbers, these guarded code parts have been triggered 736,512 times during our test, whereby 385 different code parts have been visited. To sum up, this analysis step reduced the target set from 6,745 to 385 C-specific handlers. To further reduce the candidate set, we used our debugger script to only select functions that are triggered via JavaScript. For this we assumed that any function which has a reference to part of the JavaScript engine in its call stack is valid. The survivors after this step were then manually verified.

While we focused our code path analysis on Firefox and Internet Explorer, the results of the previous analysis steps can be reused for any application. This means the static analysis of the system DLLs can be performed once and then be applied to any target program.

VI. Proof-of-Concept Exploits

To demonstrate the practical applicability of our automatically discovered crash-resistant primitives, we developed four proof-of-concept exploits that we discuss next. Note that we focussed on locating the memory oracle itself, so we assumed a memory read/write primitive to be present. During our tests we emulated such a vulnerability by modifying the target binary.

A. Internet Explorer 11

Our proof-of-concept exploit for Internet Explorer 11 relies on the function `MUTX::Enter` contained in `jscript9.dll`. It contains a call to `EnterCriticalSection` that is encapsulated in a try-catch block. The exception filter address field within the scope table contains $0x1$, which indicates that regardless of the exception code, all exceptions are caught and the execution resumes at the exception handler. The `CRITICAL_SECTION` structure passed to `EnterCriticalSection` lies within the `ScriptEngine` object at a fixed offset. The `ScriptEngine` object also contains a status field that indicates whether the last call to `EnterCriticalSection` failed. This status field is cleared before the call and set in the exception handler. The `CRITICAL_SECTION` structure contains a pointer to a `debug_info` structure. Under certain circumstances, `EnterCriticalSection` reads the field at offset $0x10$ from that `debug_info` structure. By setting three additional fields of the `CRITICAL_SECTION` structure to certain values, we can force the correct circumstances. An attacker can overwrite the pointer to `debug_info` with $x - 0x10$ to probe address x. `MUTX::Enter` is called by Internet Explorer's JavaScript engine once it processes new JavaScript code and thus, can easily be triggered by adding a new script tag to the DOM.

B. Firefox 46

As another example, we chose Firefox 46.0.1 64bit on Windows 10. As the general approach for both proof-of-concept exploits is similar, we only highlight the key differences. In contrast to Internet Explorer, where the exception handler was located in the application itself, namely `jscript9.dll`, the memory oracle in Firefox is due to an exception handler in `ntdll.dll`. While all applications import this library, the corresponding primitive was only on the execution path when using Firefox. Another difference is that the exception handler is not flagged as catch-all, instead it excludes certain exception types, but as it handles access violations it is usable for our purposes. Due to the way the memory oracle is used within the process, it does not require a manual trigger, instead a background thread continuously calls the vulnerable function. This means we only need to write the address to probe to the appropriate object and read back the result after giving the parallel thread a chance to probe.

C. Nginx 1.9

On Nginx, our framework found that the crash-resistant primitive associated to the `recv` syscall becomes available after the server receives a partial request. In detail, the server allocates a Nginx-specific `ngx_buf_t` struct object for a connection once some request data comes in (only deallocated later, when request processing completes). In our proof-of-concept exploit, we use parallel connections to implement the individual memory probes. We first send a recognizable signature via a partial request over an independent connection, so that the server allocates the buffer and saves the signature therein. While the first connection is waiting for the request to complete, i.e. for a double newline marking the end of the request, we use a second parallel connection to leak the buffer object containing our signature. Once we leak it, we perform arbitrary writes to the buffer to *reinitialize* it, i.e., set all its pointers to the memory address we are probing for. Finally, we send more data to complete a full request over the first connection. If the memory address overwritten in the buffer was inaccessible, the server gracefully closes the connection without sending back any response data. Otherwise, the server sends the requested file back to the client over the connection.

D. Cherokee 1.2

On Cherokee, our framework found a crash-resistant primitive associated to the `epoll_wait` syscall. Unlike Nginx, Cherokee's default configuration starts multiple threads to serve parallel incoming requests. Each idle thread calls the `epoll_wait` syscall in a loop, with a timeout of 1 second between iterations. Corrupting a given thread's epoll object pointer with an inaccessible memory address will cause the thread to stop serving client requests and stall in a tight loop of failing `epoll_wait` invocations. This induces a performance degradation attack on the Cherokee (lower capacity and higher overhead), resulting in a timing side channel. In our proof-of-concept exploit, we first leak the location of a given thread's `cherokee_fdpoll_epoll_t` object and then corrupt it to probe memory. For each probe, we overwrite the `struct epoll_event` pointer in the target object and measure the time for the server to handle 1,000 requests. We noticed there is significant time difference compared to the baseline when even a single thread is non-functional. With all threads running

correctly (baseline), the server handles all the requests in 5.7 seconds, and when a single thread is non-functional, it does so in 9.3 seconds (on average, with marginal variations across runs). Based on the time difference, we can distinguish whether the probed memory address is accessible (former case) or not (latter case).

VII. Discussion

In the following, we discuss limitations of our current prototype implementations and reflect on the lessons learned. We also explain the reasons for our disjoint approaches for locating memory oracles on different operating systems.

A. Locating Primitives of Previous Work

To verify our tool chain, we searched for the known memory oracles in both Internet Explorer and Firefox [22]. The primitive in IE is based on an exception handler that is set to handle all possible exceptions. As our tool looks for this kind of exception handler, we were able to locate this candidate in an automated way. After a security update, the handler handles a set of exception classes configured by a system setting. To detect this new version of the primitive, we had to manually verify it due to the filter calling another function to allow configuring the behavior.

The primitive in Firefox, on the other hand, was not located automatically because this application uses a vectored exception handler (VEH) that is registered during runtime. As we do not cover this class of handlers in our current prototype implementation, our framework can not locate this candidate. Note that this is not a fundamental limitation of our approach. Further work can support this class by locating all calls to `AddVectoredExceptionHandler` and extracting the handler address. In addition, the semantics for the symbolic execution need to be modified to account for the different function prototype.

Oikonomopoulos et al. [35] recently introduced a technique that allows an adversary to use allocations to narrow down the location of reference-less memory. While not directly related to crash-resistance, this method also provides a kind of memory oracle, but it does not rely on any fault handling. However it requires the availability of the kernel feature overcommit, which is the ability to allocate more virtual memory than is available as physical memory. We did not locate an allocation oracle as it is completely different to the primitives we targeted.

B. Differences between results on Linux and Windows

On Linux we were able to directly target syscalls and their crash-resistant nature. On Windows, certain system APIs also provide a similar behavior, but as the Windows API often contains more levels of abstraction, not all invalid arguments are passed to the system calls and, instead, result in an exception in the user-level code. However, we were still able to locate such APIs on Windows.

Besides relying on system-level functionality, a user program can also use exception handling to detect and resolve potential memory errors. Our results show that filtering prior to handling an exception can be effective in limiting the danger of a specific exception handler. Even applications that heavily use exception handling do not necessarily contain a memory oracle, if the proper filtering is performed. However we also located multiple examples of catch-all filters or filters with broad filtering criteria. Some of these were combined with memory dereferences outside of the protected code area, which usually indicates a handler which should not cover access violations, but does so anyway due to too broad filtering. Another observation is that exception handling is much more common in client applications on Windows than it is in servers for Linux. This can be explained by the differences in the way memory faults are reported on these systems. Linux uses the signal model, requiring a global signal handler to catch the corresponding signal [4], whereas Windows allows the usage of SEH, which provides a comfortable way to protect specific code blocks.

C. Potential countermeasures

Depending on the system and application different countermeasures are possible. We outline some possible defenses ranging from a system redesign to ad-hoc fixes in either the application or the system itself.

System design changes Completely eliminating memory oracles would most likely require fundamental changes to both programs and the underlying operating systems. Most of these countermeasures intentionally reduce the feature set provided to user space programs and therefore the balance between loss of functionality and gain of security must be considered. Given a mechanism to either recover from access violations or querying the state of memory addresses (either directly or indirectly), an attacker can construct crash-resistant primitives and bypass information hiding defenses. As such, we propose the following properties:

- any access violation is critical for the application and yields to termination
- exception handling can only be used for program-level exceptions (e.g., C++ exceptions), not system-level (e.g., access violations)
- error reporting and data collection is possible, but care must be taken to not allow resuming the normal execution
- system APIs and system calls must terminate the offending application on a memory error, as if the application received the fault
- exception masking must not be possible, any fault no matter the callstack terminates the process
- facilities to infer the memory layout, e.g., information in `/proc` or `VirtualQuery`, are questionable and should be removed to prevent probing using them
- allocation functions that allow specifying the desired address should be removed
- the memory layout of restarting processes must not persist between restarts

Improving exception filtering A less general approach would be to narrow the exceptions caught by specific exception handlers. This usually means that the exception filters must not

accept more than the minimal needed set of exception codes. While some handlers might need to catch access violations, their widespread use is questionable. In addition to hardening the exception handling in applications the system level oracles need to be covered as well. This could, for example, be achieved by treating a memory error within the system API or system call the same as any error in the application. This means instead of either silently discarding the exception or catching it and setting the appropriate error state, but allowing execution to continue, the application would need to explicitly employ exception handling around these functions. This results in less possibilities for an attacker to abuse such functionality, if the application can ensure no invalid pointer is passed to this API during normal operation.

Restricting access violations Another, more compatible, approach concerning memory probing itself is to only allow handling and resuming of *expected* access violations. There are two common reasons for a memory access to fail: (i) there is no mapped memory at the given address or (ii) the permissions of the memory do not allow the intended access. The first case almost always results from a wrong calculation, or in our case an exploit attempt, so it should be considered abnormal behavior. The application tried to access memory that it has not previously allocated and it also was not allocated by the operating system for the process, so there should be no references to this memory region. This is different compared to the second case where the address itself is valid—there is allocated memory at the specified location, but its permissions do not match the requested access.

Generating a fault based on permissions can be intended, it can be used for performance reasons as seen in Firefox [22], so the application explicitly allocates a region of memory, but marks it as inaccessible. In a way the program *expects* an access violation to occur under some *previously known* circumstances. As such it can be viable to only allow access violations that occur at mapped memory to be handled. This is similar to the method described by Gawlik et al. in regards to removing the scanning primitive from Firefox, but we propose to employ this policy at the system level. This means any memory access to an unmapped page causes an unrecoverable error, without invoking any exception handler in the faulting process. This can be simulated by the application itself by performing checks on the supplied exception information and terminating in the case of an unmapped access. Using this approach would still allow optimizations as used in Firefox, but scanning attempts would be detected at the first unmapped region encountered. While not providing as much security as a hard policy concerning memory errors, it reduces the odds of successful guessing significantly. This results in information hiding providing the same security guarantees as in the absence of crash resistance.

Rate based detection An orthogonal defense is a simple anomaly detection that analyzes the number of access violations. In principle, this is similar to the detection of crashes for server applications to detect a BROP attack [29]. With some applications using expected access violations for performance optimizations, we wanted to establish a baseline of how many such faults are generated during normal usage in practice. A well known instance of this design choice can be found in the Firefox web browser, so we used this program to test our theory. We added logging code to report any fault caught and handled in the browser. Using this modified version, we crawled the top 40,000 websites according to Alexa [1] and logged any occurrence. Our tests showed that none of these websites exhibited an access violation when accessing the site.

Additionally, we tested the corner case of using `asm.js`-heavy websites in the form of a dedicated `asm.js` benchmark [3]. This tool represents a stress test as it always forces native code to be generated and applies some optimizations, one of them being the usage of faults to catch out-of-bound accesses. While we observed access violations, they are far less frequent than during a probing attack similar to the one described by Gawlik et. al. [22] with multiple thousands per second. The benchmark triggered faults in groups of up to 20 in short succession, but the overall rate was much lower as there were breaks between the groups. Even if we interpret the peak rate as our baseline, the faults caused by actual scanning attempts are several orders of magnitude more frequent.

As such, we conclude that the rate of access violations can provide a viable heuristic for a defense. Even if an attacker tries to circumvent detection by performing a far slower scan, she will be slowed to a level where the duration will most likely be too high to be practical.

VIII. Conclusion

In this paper, we showed that crash-resistant primitives are not unique oddities. Most importantly, we demonstrated that memory oracles exhibit specific properties that can be used to locate them in real-world applications. We showed that it is possible to develop tools that ease the discovery of those code locations even for complex, closed-source programs. Once located, these primitives can be used by an attacker in the same way as demonstrated by previous work [22]. In addition, our results show that not only client programs are threatened by crash resistance: even servers can exhibit not only crash-tolerant behavior (as demonstrated before), but such applications are also susceptible to this new kind of vulnerabilities. Overall, our results demonstrate that locating a crash-resistant primitive is no longer left to pure chance, but poses a threat for defenses that rely on information hiding in any kind of application.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through the projects H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571, H2020 MSCA-RISE-2015 “PROTASIS” under Grant Agreement No. 690972 and ERC Starting Grant No. 640110 “BASTION”, and by the Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI “Dowsing” project.

REFERENCES

[1] Alexa - Actionable Analytics for the Web. http://www.alexa.com/.

[2] BrowserBench. http://browserbench.org/.

[3] MASSIVE - the asm.js benchmark. https://kripken.github.io/Massive/.

[4] WineHQ - Structured Exception Handling. https://www.winehq.org/docs/winedev-guide/seh.

[5] ERRNO(3) Linux Programmer's Manual. Linux Manual Page, 2016.

[6] SIGNAL(7) Linux Programmer's Manual. Linux Manual Page, 2016.

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[8] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[9] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[10] P. Betts. The case of the disappearing OnLoad exception. http://blog.paulbetts.org/index.php/2010/07/20/the-case-of-the-disappearing-onload-exception-user-mode-callback-exceptions-in-x64/.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[12] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[13] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, 2014.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, 2015.

[16] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[17] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[18] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[19] DynamoRIO contributors. DynamoRIO Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/.

[20] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[21] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.

[22] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[23] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy*, 2015.

[24] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX Security Symposium*, 2016.

[25] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, 2012.

[26] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[27] B. Kollenda, E. Göktaş, T. Blazytko, P. Koppe, R. Gawlik, R. Konoth, C. Giuffrida, H. Bos, and T. Holz. Towards Automated Discovery of Crash-Resistant Primitives in Binary Executables. Technical report, Ruhr-University Bochum, 2017.

[28] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[29] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.

[30] K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to make ASLR win the clone wars: Runtime re-randomization. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[31] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-guard: Stopping address space leakage for code reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[32] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[33] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009.

[35] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX Security Symposium*, 2016.

[36] M. Pietrek. A crash course on the depths of win32 structured exception handling. *Microsoft Systems Journal*, 1997.

[37] M. Pietrek. New vectored exception handling in Windows XP. *MSDN Magazine*, 2001.

[38] M. Prandini and M. Ramilli. Return-Oriented Programming. In *IEEE Symposium on Security and Privacy*, 2012.

[39] M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1*. Microsoft Press, 2012.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy*, 2015.

[41] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[42] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[43] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[44] Thomas Patzke. Browser Crasher. https://github.com/thomaspatzke/BrowserCrasher.

[45] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1993.

[46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.