Mitigating Data Leakage by Protecting Memory-resident Sensitive Data

Tapti Palit
Stony Brook University
tpalit@cs.stonybrook.edu

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

ABSTRACT

Gaining reliable arbitrary code execution through the exploitation of memory corruption vulnerabilities is becoming increasingly more difficult in the face of modern exploit mitigations. Facing this challenge, adversaries have started shifting their attention to data leakage attacks, which can lead to equally damaging outcomes, such as the disclosure of private keys or other sensitive data.

In this work, we present a compiler-level defense against data leakage attacks for user-space applications. Our approach strikes a balance between the manual effort required to protect sensitive application data, and the performance overhead of achieving strong data confidentiality. To that end, we require developers to simply annotate those variables holding sensitive data, after which our framework automatically transforms only the *fraction* of the entire program code that is related to sensitive data operations. We implemented this approach by extending the LLVM compiler, and used it to protect memory-resident private keys in the MbedTLS server, ssh-agent, and a Libsodium-based file signing program, as well as user passwords for Lighttpd and Memcached. Our results demonstrate the feasibility and practicality of our technique: a modest runtime overhead (e.g., 13% throughput reduction for MbedTLS) that is on par with, or better than, existing state-of-the-art memory safety approaches for selective data protection.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software Security, Data Leakage Attacks, Data Confidentiality, Side Channel Attacks

ACM Reference Format:
Tapti Palit, Fabian Monrose, and Michalis Polychronakis. 2019. Mitigating Data Leakage by Protecting Memory-resident Sensitive Data. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359815

1 INTRODUCTION

The continuous deployment of exploit mitigation technologies has made vulnerability exploitation much more challenging than it was only a decade ago [87]. It is all too telling that contestants of the first Pwn2Own competition were individual researchers who discovered vulnerabilities and wrote reliable exploits in a matter of hours [37], while the winners of recent contests comprised several teams, many of whom worked for months to develop a single exploit [38]. Besides the widespread adoption of non-executable memory pages [71] and address space layout randomization [70], the principle of least privilege is better enforced in user accounts and system services, compilers apply more protections against buffer overflows, sandboxing is increasingly used in applications that render untrusted input, and control flow integrity [10] and other exploit mitigations have become commonplace in commodity operating systems [1, 4, 31, 69, 87]. Additionally, realizing the importance of (and demand for) efficient exploit mitigations, CPU vendors have begun providing primitives that facilitate the development of lightweight and effective mitigations [19].

That said, the increasing complexity of reliably achieving arbitrary code execution, along with high-profile incidents of data leakage vulnerabilities (such as Heartbleed [3]), has prompted a renewed interest into data-only attacks [35, 60, 74, 81], which were first introduced more than a decade ago [29]. For instance, armed with an arbitrary memory access capability, adversaries can simply focus on leaking a user's HTTP session cookies for cloud storage, email, e-commerce, and other online services [74].

With the emergence of data-only attacks, protecting the data of a process, *in addition to its code*, is of paramount importance. To date, memory safety [12, 13, 27, 43, 64, 65], data flow integrity [28], data space randomization [24], privilege separation [26, 73], enclaves [42], and sandboxing [34, 45, 86, 91] have been proposed as solutions for protecting in-process data from corruption or illegal access. In practice, however, their deployment for the protection of end-user applications has been limited, due to either their high runtime overhead, or the significant code restructuring effort required. To complicate matters even more, the recent spate of microarchitectural attacks that leak secrets via side channels (e.g., Spectre [44], RIDL [84], and Fallout [59]) has aptly shown that existing in-process memory isolation technologies are not adequate for preventing sensitive data leakage.

In this paper, we propose a practical approach for countering data leakage attacks against user-space applications. The core idea stems from the observation that, depending on the application, some data is more critical than others. By focusing only on a subset of data, we can achieve a low-enough runtime overhead by amortizing the cost of the protection mechanism, while offering strong data confidentiality. Sensitive data is always kept encrypted in memory, and is decrypted only while being loaded into registers for carrying out computations. Similarly, the secret key state used to encrypt and decrypt sensitive data is always stored only in registers, and in particular in the AVX2 [9] registers that have been available since 2013 (introduced in the Intel Haswell architecture). Consequently, even if attackers can repeatedly read arbitrary memory (e.g., by exercising an arbitrary read primitive through malicious JavaScript code), any leaked sensitive data will always be encrypted.

We implemented our solution on top of the LLVM compiler, and rely on whole-program pointer and data flow analysis at the LLVM-IR level to pinpoint all the code points that access sensitive data, and instrument them appropriately. A core design goal is to minimize the effort needed to protect an application, by requiring developers to just *annotate* only any *initial* sensitive data or data sources (e.g., cryptographic keys, passwords, HTTP session cookies) without the need for further source code modifications. Sensitive data is often not heavily propagated, thus limiting the performance overhead associated with cryptographic operations. As such, we can protect sensitive data with limited program instrumentation.

We empirically assess the practicality of our technique using a set of microbenchmarks and real applications. Our results show that the runtime overhead is modest (e.g., 13% throughput reduction for the MbedTLS SSL server when protecting its private key), achieving performance that is on par with or better than existing state-of-the-art memory safety approaches for selective data protection [27]. An additional benefit compared to existing memory safety and data isolation approaches is that it offers protection against recent microarchitectural attacks that rely on speculative execution [44], as any leaked data always remain encrypted. At the same time, our work highlights important challenges in the front of whole-program fine-grained pointer analysis that leave room for significant improvement once resolved.

Our work makes the following main contributions:

- We propose a compiler-level defense against sensitive data leakage attacks for user-space applications. Using whole-program pointer and data flow analysis, our technique instruments only the fraction of the program code needed to keep sensitive data always encrypted in memory.
- An implementation on top of LLVM that requires only minimal developer intervention in the form of simple code annotations to protect the confidentiality of sensitive application data.
- An in-depth assessment that shows that we can achieve our goals with modest runtime overhead.
- An evaluation against a publicly available Spectre proof-of-concept attack, which demonstrates how our approach protects sensitive data against microarchitectural side-channel attacks.

2 BACKGROUND AND MOTIVATION

After a decade-long hiatus since the introduction of data-only attacks [29], several advancements that demonstrate their power have been brought to light [35, 40, 41, 60, 61, 74, 81]. These works take advantage of memory disclosure vulnerabilities to access arbitrary memory and subsequently provide adversaries with powerful capabilities [15, 22, 46, 48, 52, 77]. Heartbleed [3] is a recent example that demonstrates how the ability to read arbitrary memory can be used to leak sensitive application data, such as private keys.

To protect sensitive in-memory data from leakage, it is thus important to consider the adversarial capabilities enabled by memory disclosure vulnerabilities, especially when combined with scripting support [74]. Unfortunately, application sandboxing protections (or sandboxing policies enforced through SFI [86], XFI [34], or data sandboxing [91]) cannot protect against these attacks, as data leakage still occurs within the enforced boundaries. On the other hand, stricter data isolation policies, such as data flow integrity (DFI) [28] do protect against data-only attacks, but incur a prohibitively high runtime overhead (e.g., 104% for the SPEC benchmarks).

Another mitigation against data-only attacks is to change the representation of in-memory data, by always keeping it transformed and restoring its original representation only when it needs to take part in some computation. As an initial exploration of this idea, Bhatkar and Sekar [24] proposed an approach for XOR-ing data objects with a random per-object "key" that is kept alongside each object in memory. Under the stronger disclosure-aided exploitation threat model, however, this form of data space randomization does not offer adequate protection, as the key cannot be kept secret. Moreover, the runtime overhead—due to the necessity of XOR operations before and after each and every memory access to each and every object—is prohibitively high.

In this work, we revisit the idea of data space randomization, but with the goal of achieving stronger protection even under arbitrary memory read capabilities. As simple XOR-ing can be defeated by comparing known data with its transformed version, we use stronger encryption without introducing substantial computational overhead [58]. To that end, we leverage the AES-NI instruction set extensions for hardware-accelerated AES computations, along with the AVX2 [9] registers for storing the expanded round keys for each AES operation.

In comparison to existing memory safety and data flow integrity approaches, which instrument the *entire* program to prevent arbitrary access to the protected data, our sensitive data protection approach instruments only the *fraction* of instructions involved in sensitive data flows and operations, and ignores the rest of the memory-related instructions—these may still illegally access the protected data, but only in its encrypted form.

In comparison to existing approaches based on privilege separation [26, 73], hardware-based protection [36, 62, 85] or enclave solutions like SGX [17, 25, 53, 76, 82], our approach does not require any code refactoring or rewriting, besides a simple annotation of existing data variables or data sources.

3 THREAT MODEL

We consider the broad class of memory disclosure or corruption vulnerabilities that give adversaries the capability to read (i.e., leak) arbitrary user-space memory. We assume that due to the nature of the vulnerability (e.g., as was the case with Heartbleed [3]), or due to the deployment of exploit mitigation mechanisms, immediate arbitrary code execution is not possible, and thus the adversary is constrained in mounting some form of data-leakage attack. The attack may be facilitated by the execution of malicious script code that leverages the disclosure vulnerability to repeatedly access arbitrary memory [74]. Because adversaries do not have arbitrary code execution capabilities, however, they cannot disclose the sensitive data and the expanded round keys stored in registers.

Although the end goal of some advanced data-only attacks is to *modify* configuration or control data [35, 61], our approach is tailored to defending against data *leakage* attacks, which still comprise an important sub-class of data-only attacks [3, 74]. In this work, we focus on maintaining the confidentiality of sensitive data, but the integrity of such data may not be fully protected. Specifically, the encryption scheme we utilize offers some level of protection against data modification attacks, but cannot prevent certain attacks that rely on replacing data with other already encrypted values. We discuss in detail such attacks, along with the challenges of fully guaranteeing data integrity, in Section 7.

We focus on the protection of user-space applications, and thus assume that adversaries do not have access to any kernel-level code or data. Nonetheless, we assume that the attacker can perform cold boot attacks. Because all sensitive data is present in RAM in encrypted form, and the secret round keys are present only in registers, the attacker can not recover the plaintext by simply reading the physical memory.

With respect to the recent wave of CPU side channel attacks that allow arbitrary memory access from user space, our solution does not protect against Meltdown [54], as protecting kernel attacks is out of scope. However, it *does offer* effective protection against Spectre [44] and similar microarchitectural attacks based on speculative execution. Spectre attacks leak arbitrary data that has been loaded into the cache within the scope of a user-space process. Thus, these attacks will access protected data *only in its AES-encrypted form.*

4 DESIGN

The proposed approach aims to strike a balance between the manual effort required to enable the protection of sensitive application data, and the performance overhead of the data protection mechanism itself. Existing application-level isolation technologies such as privilege separation [26, 73], enclaves [42], and sandboxing [34, 86, 91], have a relatively low performance impact, but require an immense code refactoring effort. As part of this process, one must identify and move the sensitive data (and all associated critical-path code) into the protected domain, and implement appropriate interfaces with the rest of the application code.

By contrast, we merely require developers to *annotate* sensitive data in the source code, without requiring any further code modifications. Before any computation is performed, the data is first decrypted and stored in a register, which is the only location in which plaintext sensitive data is ever exposed. A decryption “boundary” is defined at the system call level, to allow for seamless interaction with the OS or inter-process communication by supplying decrypted data to domains outside the reach of an attacker. To achieve these capabilities, several challenges must be addressed:

(1) The whole code of the process must be considered, including the main application and all its libraries.
(2) All pointers that may reference a sensitive object must be identified and handled accordingly.
(3) Data marked as sensitive may propagate to other (non-marked) variables and objects.
(4) The unit of encryption for AES is 128 bits, but sensitive data objects may be smaller or larger than that.

Our design is centered around addressing the above challenges. In the rest of this section, we describe the different types of analysis and code transformation required for protecting a given application.

4.1 Whole-Program Analysis

To ensure that sensitive values are never left decrypted in memory, our approach must analyze and transform the whole program code, including any external libraries, because sensitive data might be passed as arguments to functions in these external libraries. This requires the source code of the application and all dependent libraries to be available for analysis and transformation.

Performing whole-program analysis at the source code level is difficult, as merging the source code of different libraries may result in clashes due to identically named static functions and variables. To avoid these issues, we opt for merging the code object files after LLVM transforms them to their intermediate representation (IR), at which point any identically named static functions are automatically renamed. Moreover, operating at the IR level gives us access to LLVM’s sophisticated analysis and transformation capabilities available at this level. Also, ensuring that the sensitive data remains protected through the LLVM backend passes requires interfacing with them, and operating at the IR level makes this easier.

Link time optimization gives LLVM the capability of dumping the IR of a compilation unit on disk. This allows the IR of multiple compilation units to be optimized as a single module. The LLVM toolchain provides the necessary tools to generate static libraries from these IR units, thus allowing link time optimization of the application along with its library dependencies.

4.2 Pointer Analysis

Once the programmer annotates an object or a variable as sensitive, every valid access to these objects must be transformed with the appropriate encryption or decryption routines. Given the heavy reliance of C and C++ code on the use of pointers, we must first determine which pointers may hold references to sensitive objects, so that the respective pointer dereference operations can be also transformed accordingly. To that end, as part of the static analysis performed at the IR level, we employ *pointer analysis* to resolve all possible memory objects that a pointer might refer to.

4.2.1 Sensitive Data Domain. The LLVM optimization phase already provides implementations of various pointer analysis algorithms. However, these implementations support only intra-procedural analysis capabilities, which are not adequate for our purposes. Instead, we use an inter-procedural version of Andersen’s algorithm [14]. This well-known flow-insensitive pointer analysis algorithm examines pointer-related statements one by one, and updates a *points-to* graph with any newly found points-to relationships. Each node of the graph represents either a pointer or a memory object, and each edge represents a points-to relationship.

Figure 1 shows a small C code example and its corresponding points-to graph. The points-to set of pointer `ptr1` includes the variables `a`, `b`, and the array `arr`, but only variable `a` has been annotated as sensitive. Because `ptr1` can point to any of the objects in its points-to set, we must treat all three variables as sensitive.

```
1  void fun1(void) {
2      SENSITIVE int a;
3      int b, c;
4      int arr[10];
5      int *ptr1, *ptr2;
6
7      ptr1 = &a;
8      ptr1 = &b;
9      ptr2 = &b;
10     ptr2 = &c;
11
12     for (int i = 0; i < 10; i++) {
13         ptr1 = &arr[i]; ...
14     }
15 }
```

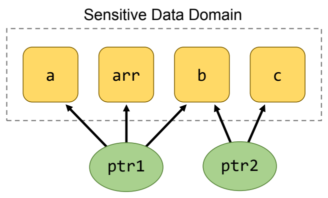

Figure 1: Example C code with an integer variable marked as sensitive (line 2), and the corresponding points-to graph.

Moreover, once we mark variable b as sensitive, ptr2 (which points to b) must also be marked as sensitive, and in turn, variable c becomes sensitive as well.

These relationships form an equivalence class of sensitive data, which we call the *sensitive data domain*, depicted in the upper part of the points-to graph. This example illustrates one of the major challenges we faced—that is, the results of pointer analysis are in general an over-approximation of the actual relationships among objects, which consequently results in an over-approximation of the actual sensitive data domain. Ideally, we would like our analysis to have maximum precision to minimize the instrumentation overhead. Unfortunately, higher degrees of precision usually entail longer computation time for the analysis, and in certain cases, may give rise to other challenges specific to our use case.

4.2.2 Field Sensitivity. Field sensitivity [18, 72] is an approach for improving the precision of pointer analysis, and refers to the ability of the analysis algorithm to distinguish between individual fields of a complex object, such as a C struct. This is particularly important in case of complex objects containing multiple pointers that may point to distinct sets of objects in memory. Unlike field-insensitive analysis, field-sensitive analysis treats each of these pointers (of the same complex object type) as distinct. Field-sensitive pointer analysis is thus more precise than field-insensitive analysis, and would result in a smaller sensitive data domain.

Using field-sensitive analysis for protecting sensitive data is by no means an easy feat. Numerous challenges abound. For one, while the block size for AES operations is 128 bits, the individual fields of a struct object will often not be aligned at 128-bit boundaries, requiring extra padding and alignment. We describe this and other related challenges in detail in Sections 4.4 and 5.1.3. As shown by our experimental evaluation, switching to field-sensitive analysis resulted in a considerable reduction of the overall runtime overhead compared to field-insensitive analysis.

4.3 Value Flow Analysis

Resolving all pointer references is not enough to achieve complete data protection, as sensitive data may propagate to other variables and objects, which we call *sensitive sink sites*. To prevent potential information leakage through them, we use value flow analysis to recursively find all such sensitive sink sites. All memory accesses to these sites are then instrumented with appropriate encryption or decryption transformations.

To correctly track sensitive value flows through function calls, we first resolve the targets of function pointers using the information generated from the prior pointer analysis phase, which allows for the creation of a sound call graph. Having the call graph, we can then track sensitive values passed as arguments to other functions, as well as any sensitive values returned by functions. Sensitive value flows can be direct or indirect. Indirect flows occur due to the presence of pointers. Due to the reliance on the prior pointer analysis phase for resolving the targets of pointers, our value flow analysis is also affected by the precision of the pointer analysis.

The combination of pointer and value flow analysis gives us the full set of sensitive data objects that must be kept encrypted in memory, and the corresponding code instrumentation points.

4.4 In-Memory Data Protection

Once all memory objects in the sensitive data domain have been discovered, as a result of the pointer and value flow analysis phases, the final step is to instrument the respective memory read and write operations with calls to custom decryption and encryption routines. We opted for the strong data confidentiality that AES [30] offers, to avoid the risk of cryptanalysis-based attacks that an adversary could mount through script code (or even offline). Modern processors offer native support for accelerating AES operations, e.g., as is the case with Intel's AES-NI extensions [39].

A major engineering challenge we faced stems from the fact that the basic unit of operation for AES is 128 bits, but sensitive scalar values may be 8, 16, 32, or 64 bits in length, while data objects such as private keys, passwords, and configuration-related data structures, are often larger than 128 bits. The frequent size mismatch between objects and AES block size prevents us from applying AES directly to protect individual objects. Dealing with smaller objects is relatively straightforward by padding them to 128 bits, although this entails several implementation considerations for different types of memory (global, stack, heap), which we discuss in Section A.1 of the appendix. On the other hand, dealing with larger objects unavoidably requires processing them in 128-bit blocks. In both cases, objects are 128-bit aligned to optimize memory offset computations.

Decrypted Data Cache. To optimize the common case of repeated accesses to the same data, we implemented a *decrypted data cache* to minimize the number of cryptographic operations over time for a given block. Our requirement of never exposing plaintext sensitive data in memory explicitly rules out the possibility of using any memory-resident buffer for this purpose. However, we can take advantage of spare CPU registers to temporarily hold decrypted data—leaking register contents requires the execution of arbitrary (i.e., non-instrumented) code, which (based on our threat model, discussed in Section 2) falls outside the attacker's capabilities.

The x86 Streaming SIMD Extensions provide support for 16 128-bit registers (named XMM0 to XMM15) in 64-bit processors (or eight 128-bit registers in 32-bit processors). When accessing a sensitive value from memory, we first decrypts the 128-bit block that contains the sensitive value, and loads it into the XMM0 register. In case of a read operation, the respective byte/word/double-word is copied from the XMM0 register to the required general purpose register—after that point, all arithmetic or logical instructions that follow the memory read proceed unchanged. In case of a write operation, the new value of the required byte/word/double-word is written in the appropriate offset in the XMM0 register.

Instead of immediately clearing the XMM0 register, the decrypted contents are retained for as long as possible. Any subsequent access to the same block can be directly accommodated from the already decrypted contents of the XMM0 register. When a subsequent sensitive memory operation accesses a different 128-bit block, the current block is re-encrypted and written back to memory before proceeding. The register is also re-encrypted and written back before calls to any external interface. This simplified caching approach takes advantage of the locality of data accesses to reduce the overhead of repeated AES operations on the same data.

5 IMPLEMENTATION

The Clang frontend translates C/C++ code to the LLVM intermediate representation, which is then lowered into assembly by the LLVM backend. LLVM provides a powerful and expressive framework for analysis and transformation at the IR level, and thus most of our implementation was performed at that level. The LLVM IR also simplifies the high level C/C++ code to enable efficient code transformations and analysis.

The LLVM compiler toolchain is modularized into several passes, with most of the passes operating at the IR level. Each pass carries out a single *analysis* or *transformation* task. We implemented pointer analysis and value flow analysis as two separate analysis passes, and the final AES instrumentation as a transformation pass. Figure 2 illustrates how the different phases are integrated into the LLVM toolchain.

The Clang frontend lowers the `SENSITIVE` annotation to a call to the `llvm.var.annotation` function, which takes as arguments the objects that were annotated as sensitive. We first collect these arguments to find the initial set of sensitive objects. Then, at the IR level, this set of objects becomes the starting point of our analysis and transformation passes.

5.1 Link Time Optimization

We modified LLVM to invoke our analysis and transformation passes during the LTO phase, which enables us to support static libraries and standalone applications. This also has the additional benefit of not requiring any modifications to Makefiles, except for passing custom values to environment variables, such as `CC`, `AR`, `RANLIB`, and `CFLAGS`.

5.1.1 Pointer Analysis. To perform whole-program analysis, we extended the the Static Value Flow (SVF) analysis framework [80], which supports pointer analysis and program dependence analysis for C and C++ programs. SVF first analyzes the LLVM IR instructions in the merged IR and gathers constraints that model the flow of pointers in the program. These constraints are represented in the form of a *constraint graph.* Then, using an inter-procedural Andersen's style pointer analysis algorithm [14], SVF iteratively performs pointer analysis by performing a reaching analysis on this constraint graph, followed by call graph construction. Each iteration of pointer analysis may discover new function pointer targets, and therefore updates the call graph with new call edges. Each new edge in the call graph may expose new pointer flows, thus requiring the pointer analysis to be repeated. This iterative execution continues until no new edges are added to the graph, i.e., until reaching a "fixed point."

SVF ensures that the result of the pointer analysis is sound. The pointer analysis provided by SVF is field-sensitive. As discussed in Section 5.1.3, field-sensitive analysis results in individual fields of a structure becoming sensitive. This causes problems because the AES unit of encryption is 128 bit, and individual fields are often neither aligned to 128 bits, nor 128 bit wide. Therefore, in addition to the field-sensitive version that handles these complex struct-field alignment cases, we also implemented a simpler field-insensitive version. For this, we modified the processing of constraints so that accesses to individual fields of complex objects are treated as accesses to the entire object. As discussed in Section 4.2.2, this field-insensitive pointer analysis results in an over-approximated sensitive data domain, but provides a simpler implementation alternative and does not require widening and aligning of the individual fields of structures.

Using the results of the pointer analysis, we populate two maps: `pointsToMap`, which maps pointers to their possible targets, and `pointsFromMap`, which maps objects to pointers that may point to them. Once the results of the pointer analysis and the value flow analysis are available, we construct the equivalence class for the sensitive pointers and objects. The pseudo-code for this process is provided in Algorithm 1 in the appendix.

5.1.2 Value Flow Analysis. As discussed in Section 4.3, data of objects marked as sensitive may be copied and stored to other objects (sink sites). Given that these objects must remain encrypted in memory, we perform interprocedural value flow analysis to find them and instrument them appropriately.

The LLVM instructions `LoadInst` and `StoreInst` are used to read from and write to memory, respectively. For the purposes of our value flow analysis, we track the flows that begin from a `LoadInst` reading a sensitive object, and terminate in a `StoreInst` writing to a non-sensitive object. As discussed earlier, SVF represents the constraints required for points-to analysis in the form of a constraint graph, which it then solves to resolve the targets of every pointer in the program. We leverage this graph, and add edges corresponding to the value flows caused by `LoadInst` and `StoreInst`. After the points-to analysis is complete, we perform a breadth-first graph traversal to derive the sensitive sink sites. Because the solution of the SVF constraint graph contains the targets of function pointers, we can trivially track inter-procedural value flows even in the presence of function pointers.

The results of the pointer analysis are used to track indirect sensitive value flows through pointers. We first find which pointers might point to sensitive objects. Then, we perform value flow analysis on the values defined by `LoadInst` instructions that perform an indirect memory read using these pointers.

5.1.3 Partially Sensitive Complex Objects. Field-sensitive points-to and value flow analysis may cause *individual* fields of struct-type objects to become sensitive. This creates a problem because these individual fields are often smaller than 128-bits long. One

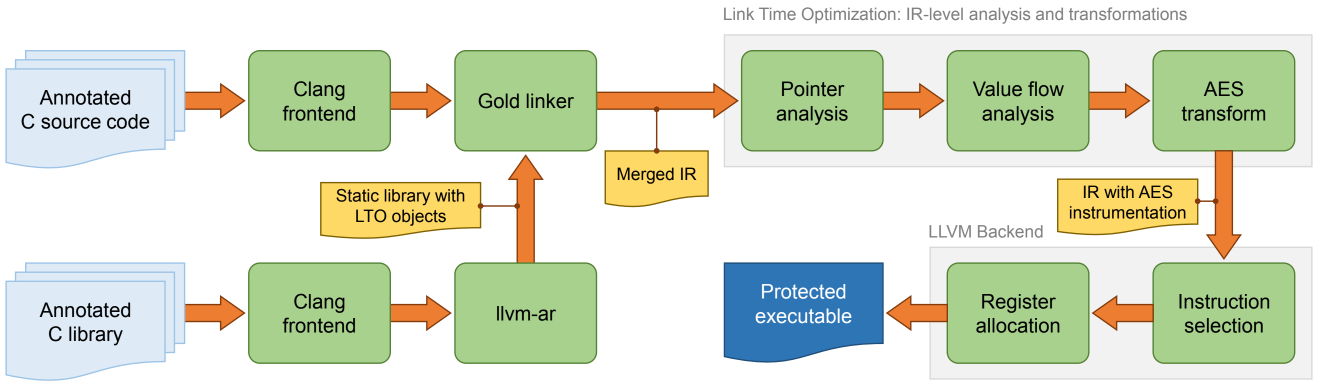

Figure 2: Overview of our sensitive data protection approach as implemented in LLVM.

solution could be to align all fields of such a partially-sensitive struct-type objects to 128 bits, but this has the risk of degrading cache performance, as the individual fields of the objects would be spaced further away in memory. To minimize performance impact, we align only those fields of a struct that are sensitive to 128-bit boundaries, and also pad them to 128-bit size.

A second challenge is the use of the `sizeof` operator, which allows the programmer to retrieve the allocation size of an object in memory. This operator is lowered by the Clang frontend into constants according to the object size, *before* the IR is constructed. With our approach, however, the correct size of partially sensitive objects becomes available only *after* alignment and padding, which is performed at the IR level. We address this issue by modifying the Clang frontend to append each instruction that uses a `sizeof` operator with custom metadata (passed on to the IR level) that includes the struct type on which the `sizeof` operator was applied. Once the points-to and value flow analysis have completed, we revisit these instructions and recalculate the sizes of any struct-type objects based on the alignment and padding of their sensitive fields. We then fix up the constants corresponding to the `sizeof` operators with the recalculated sizes.

5.1.4 *Memory Encryption Transformations.*

The Sensitive Data Domain contains the set of memory objects that must be kept encrypted in memory. This set includes global objects, objects on the heap, and objects on the stack, which in LLVM IR are represented by `GlobalVariable`, `CallInst`, and `AllocaInst` class objects, respectively. First, we use the `pointsFromMap` provided by the pointer analysis to find all sensitive pointers that might refer to these objects. Then, we collect all `LoadInst` and `StoreInst` instructions that read and write to sensitive objects in memory, either directly or via sensitive pointers. These instructions must be rewritten to decrypt or encrypt the sensitive objects. We present the details of this code transformation phase in Section 5.1.5 below.

To apply these cryptographic transformations, objects must be 128-bit aligned, while global variables with default initializers, constant values, and environment variables within the sensitive data domain, must be initialized to the correct encrypted values. We handle these special cases by adding the correct transformations to the IR. To ensure that sensitive values read from memory remain protected during the subsequent stages of the compilation process, we add a `SENSITIVE` metadata tag to the values defined by the `LoadInst` instructions, which is propagated to the LLVM backend.

5.1.5 *Hardware-accelerated AES and Key Protection.*

Intel processors provide the `aesenc`, `aesenclast`, `aesdec`, and `aesdeclast` instructions (as part of the AES-NI extensions) to speed up AES operations. The latest Intel processors also support the Streaming SIMD Extensions (SSE) [6] and the more recent, Advanced Vector Extensions (AVX) [56]. Intel SSE provides 32 128-bit registers (XMM0–XMM15), and AVX widens them to 256 bits (YMM0—YMM15). Intel SSE also includes instructions for writing and reading individual 8/16/32/64 bytes from XMM registers (`pinsrb`, `pinsrw`, `pinsrd`, `pinsrq`, and `pextrb`, `pextrw`, `pextrd`, `pextrq`, respectively). Similarly, AVX includes instructions for reading individual 128-bit chunks from the YMM registers (`vinserti128` and `vextracti128`).

We use these instructions to perform cryptographic operations oblivious to memory leakage. Using a 128-bit key with AES requires 10 processing rounds, each consuming four words (128 bits) from the key schedule (derived from the initial 128-bit key), also referred to as "round keys." Before any round-based processing begins, the input value is XOR-ed with the first four words of the key schedule (for a total of 11 four-word keys). To avoid the overhead of generating the round keys from scratch before each AES operation, they should ideally be pre-generated from the initial secret key, and stored in registers [45, 58]. To protect these round keys from memory disclosure vulnerabilities, the code that loads them into registers is placed on its own 4KB page, which is zeroed out immediately upon its execution.

Storing all the round keys in registers would require 22 128-bit registers. Processors with AVX support provide access to 16 256-bit registers, which can be accessed independently as 32 128-bit registers. However, Libc and other libraries rely on XMM registers to perform optimizations such as loop unrolling. To maintain compatibility with such optimizations, we use only the 15 YMM registers to store *all* ten expanded encryption round keys, a subset (four) of the expanded decryption round keys, and the single XOR key. Decryption round keys are the inverse of the encryption round keys, and Intel provides the `aesimc` instruction to compute the decryption round key, given its encryption counterpart. Per Intel's documentation, we use this instruction to compute the remaining six decryption round keys on the fly as needed.

As noted earlier (Section 4.4), we use the 128-bit XMM0 register as our decrypted data cache. We use the SSE instructions to read or write individual values in an already decrypted block stored in XMM0, and load the AES round keys into the XMM1 register.

The logic for loading the keys into registers is encapsulated in a function named `populate_keys`. To effortlessly rotate the keys upon each new program invocation, we rely on the binary analysis and rewriting capabilities of Pyelftools [21], which we used to implement a custom program that replaces all instances of the old encryption and decryption keys with new user-provided (or randomly generated) values.

5.1.6 Handling Common Libc Functions. Functions such as `strcpy`, `strlen`, `strcmp`, and their memory counterparts `memcpy`, `memcpy`, and `memset`, are utility functions that are invoked with a variety of arguments. Some of these arguments are sensitive, while others are not. If we were to mark the arguments to these functions as sensitive, then any invocation with even a single sensitive argument would require the other non-sensitive arguments to also be included in the Sensitive Data Domain, as discussed in Section 4.2.1. This would cause these other arguments to also be marked as sensitive and be encrypted in memory. This would increase the performance overhead as they would have to be decrypted to be computed on.

We solve this challenge by providing custom sensitive and non-sensitive implementations of these commonly used functions. For example, if the Libc function `strlen` is invoked at two places, once with a sensitive string, and once with a non-sensitive string, the first instance will invoke the sensitive implementation of `strlen`. This version decrypts every byte of the string, as it checks for the NULL termination character. The other invocation will invoke the vanilla implementation of `strlen`. This approach prevents the over-approximation of the Sensitive Data Domain, and the resulting additional performance overhead.

5.2 LLVM Backend

The LLVM backend lowers the IR to assembly code. We propagate the sensitive metadata associated with every sensitive IR value through the different phases of this lowering process. Instruction selection and register allocation are two critical phases of this lowering step. Using the sensitive metadata, we made a number of modifications to these phases to guarantee that sensitive data remains encrypted in memory. We discuss each in turn.

5.2.1 Instruction Selection. One of the requirements of encrypting sensitive data in memory is that no instruction can directly operate on in-memory operands. However, the x86 architecture supports in-memory operands for arithmetic and logical instructions. Directly accessing in-memory encrypted operands, without decrypting and storing them in registers first, will give incorrect results for the operation. Based on our experimentation, we observed that LLVM's `FastISel` instruction selection algorithm prefers the selection of instructions with in-register operands, over those with in-memory operands. However, to ensure the absolute correctness of our implementation, we modified `FastISel` to select arithmetic and logical instructions with solely in-register operands.

5.2.2 Register Allocation. Registers in LLVM's IR are virtual and infinite. As the IR is lowered to architecture-specific instructions, virtual registers are mapped to physical architecture-specific registers. Due to the limited number of physical registers, values stored in them may be spilled to memory. We use the metadata collected during the memory encryption transformation (described in Section 5.1.4), to track the virtual registers that contain sensitive values.

LLVM's `FastRegAlloc` register allocation algorithm maps each virtual register to a slot on the stack. When register pressure increases, it selects a virtual register to spill on the respective stack slot. We modified LLVM's `FastRegAlloc` to encrypt the values stored in sensitive virtual registers before spilling them their designated stack slots, and re-encrypt them when they are restored.

6 EXPERIMENTAL EVALUATION

To investigate the performance overhead of the proposed approach, we evaluated our prototype with stress-test microbenchmarks and five real-world applications. In the microbenchmarks, we annotate all data used for computation as sensitive, whereas in the real-world applications, we mark only data that is critical from a security perspective as sensitive. To illustrate the impact of pointer analysis accuracy on performance, in case of the real-world applications, we evaluate both our simpler field-insensitive implementation, as well as the more fine-grained field-sensitive implementation. Note that pointer analysis accuracy does not have an impact on the microbenchmarks, in which all data is marked as sensitive. In addition, we performed experiments to verify that sensitive data always remain encrypted in memory, and to demonstrate how this thwarts Spectre attacks.

Our testbed consists of a server with an Intel Xeon E3-1240 v6 processor, and a client with an Intel Xeon E5-2620 v4 processor. Both machines run Ubuntu 16.04.3 LTS, and use Glibc version 2.23. Single-machine benchmarks were run on the server machine.

6.1 Microbenchmarks

As discussed in Section 2, previous works on data space randomization [24] rely on XOR-based transformation to protect *all* in-memory data. In the face of memory leakage vulnerabilities, however, strong encryption must be used to ensure data confidentiality. Unfortunately, unrestrictedly encrypting all data in memory results in a prohibitively high runtime overhead, which we set out to explore with a pair of worst-case microbenchmarks.

The first program computes the sum of ten billion randomly generated 64-bit integers, which are stored in a dynamically allocated buffer that is annotated as sensitive. We measured the average CPU user time to compute the sum across multiple repetitions, which resulted in a runtime overhead of 390%. By inspecting the output of the the value flow and pointer analysis, we observed that 95% of all memory read and write operations across the whole code access sensitive memory regions. These accesses are the main source of the runtime overhead due to cryptographic operations.

The second program uses the quicksort algorithm on ten billion randomly generated 64-bit integers. The key difference of this benchmark from the previous one is that its memory access pattern is more random. We observe that close to 96% of all memory reads and writes access sensitive memory regions. However, due to the

Table 1: Fraction of instrumented instructions among all memory-related instructions in the code, and all memory-related instructions executed.

Application	Code		Execution	
	Field Ins.	Field Sen.	Field Ins.	Field Sen.
MbedTLS SSL Server	31%	11%	26%	15%
Lighttpd with ModAuth	20%	5%	26%	11%
Memcached with Auth.	0.1%	0.1%	~0%	~0%
ssh-agent	17%	8%	8%	3%
Minisign	28%	14%	55%	27%

random memory access pattern, fewer accesses can be served from the already decrypted contents in the XMM0 register, which resulted in a higher overhead of 650%. These results clearly motivate the need for protecting only a subset of the data.

6.2 Applications

The test cases of benchmark suites typically used for performance evaluation, such as SPEC2006 [2], do not involve data that is clearly sensitive. Moreover, our microbenchmark experiments show that the cost of encrypting all data in a process using AES is prohibitively high. To assess the overhead of our approach under realistic conditions, we evaluate our implementation using five real-world applications and libraries. We opted for a diverse set of both server (MbedTLS, Lighttpd, Memcached) and client (ssh-agent, Minisign) applications that handle critical user data, such as secret keys and passwords. The size and complexity of these applications is adequate for our current static analysis capabilities, and is on par with what other alternative selective data protection solutions can support (e.g., DataShield [27]).

6.2.1 MbedTLS Server. Our first application is the `ssl_server2` server that comes with MbedTLS [7], an SSL/TLS library written in C. We built a minimal version of the MbedTLS library, including only the RSA and AES ciphersuites. Our modified LLVM toolchain does not support inline assembly yet, so we disabled the use of inline assembly in the MbedTLS configuration options.

The private key of the SSL server is stored in an object of type `mbedtls_pk_context`, which we annotate as sensitive. This is the *only* manual step involved—our LLVM-LTO toolchain then automatically generates a merged IR object file, which comprises both the SSL server and the MbedTLS library, and performs value flow and pointer analysis to find and instrument all memory operations that access sensitive data. In Table 1, we report both the number of memory-related instructions that are instrumented in the code, and the number of instrumented memory accesses executed at runtime. Across all memory accesses, only 31%, for the naive field-insensitive approach, and 11%, for the field-sensitive approach involved sensitive memory objects, and thus had to be instrumented.

We deployed the instrumented server and the unmodified `ssl_client2` program on the server and client machines, respectively. The client makes 500,000 consecutive requests to the server, with each request fetching the same 200 byte HTML page. Table 2 shows the performance overhead incurred by the instrumentation. When the field-insensitive analysis is used, the instrumentation reduces the throughput by 28%. Although the performance overhead is higher than one would want in practice, the main culprit is the imprecision of the field-insensitive pointer analysis algorithm, which over-approximates the sensitive data domain that is protected.

When switching to the field-sensitive implementation, the overhead is limited to only 13%, regaining the performance that was lost due to field insensitivity. As a comparison data point, Carr and Payer [27] reported a 35.7% overhead for a similar experiment of applying DataShield on `ssl_server2`.

6.2.2 Lighttpd with ModAuth. Lighttpd is a popular, lightweight web server. Lighttpd's `ModAuth` module supports HTTP Basic Access Authentication, a method for an HTTP user agent to provide a username and password while making a request, which are stored in a preconfigured file on the server. The password is loaded from this file to the variable `password_buf`. We annotated this variable as sensitive and compiled the server using our framework. Using the hardened binary, we performed 2,000 requests to a password-protected 1 KB web page. In case of the field-insensitive approach, the throughput degrades by 22%, and for the field-sensitive approach, the throughput is reduced by 8%.

6.2.3 Memcached: Authentication using SASL. Memcached is a popular in-memory key-value store, used to improve web server performance by caching the results of expensive database queries. Memcached provides an authentication mechanism that can be used to deploy it in untrusted networks, which relies on the SASL (Simple Authentication and Security Layer) library.

For simple password-based authentication, the function `sasl_server_userdb_checkpass` loads the password from the specified password file and stores it in the `buffer` variable, which we annotate as sensitive. We use the hardened binary to perform 1M "set" and "get" operations, which store and retrieve keys in the Memcached server, respectively. Because the authentication step is performed once at the time of connection establishment, each operation is performed over a new connection and is preceded by an authentication step. Our results show that for both approaches, the throughput overhead of our instrumentation is negligible. This is because there is only one pointer to the stored password, and the password is not copied to any other memory location. Moreover, this pointer is not part of any complex C struct, and thus both the field-insensitive and field-sensitive approaches give the same results. Also, the code that checks for password validity accesses the password sequentially, maximizing the use of the AES cache.

6.2.4 `ssh-agent`. The `ssh-agent` daemon holds a user's decrypted private keys in memory to speed up the creation of new SSH sessions, by avoiding having to type the key's passphrase. Applications such as `ssh`, `scp`, and `git`, which require access to the user's decrypted private keys, communicate with `ssh-agent` over a Unix domain socket to carry out the SSH authentication process.

To reduce dependencies on external libraries, we built `ssh-agent` with support only for the internal crypto engine. When a user adds a new private key, `ssh-agent` dynamically allocates an `sshkey` object on the heap. We annotate the pointer returned by this allocation

Table 2: Performance evaluation results. Overhead numbers correspond to throughput for the first three servers, and user time for the last two programs.

Application	**Run-time**	**Run-time**		**Overhead**	
	(original)	**Field Ins.**	**Field Sen.**	**Field Ins.**	**Field Sen.**
MbedTLS SSL server (500,000 requests)	110s	152s	126s	28%	13%
Lighttpd with ModAuth (2,000 requests)	37s	47s	40s	22%	8%
Memcached with Auth. (1M Get/Set req.)	67s	67s	67s	0%	0%
ssh-agent (2,000 user sessions)	450s	485s	469s	8%	4%
Minisign (1GB file signing)	41s	69s	54s	68%	33%

as sensitive. This ensures that all private keys in dynamically allocated `sshkey` objects always remain encrypted in memory. Based on our IR-level static analysis results, 17% (field-insensitive) and 8% (field-sensitive) of all memory operations required instrumentation, while 8% and 3% of all memory operations performed at runtime were instrumented, respectively.

Using the same setup, we deployed the instrumented `ssh-agent` daemon on the client machine and set it up with the user's private keys. Public key authentication to the server machine was preconfigured. The experiment consists of the client making 2K logins to the server. We measured the total time taken for the 2K logins, and report an overhead of 8% (field-ins.) and 4% (field-sen.).

6.2.5 Minisign: File Signing using Libsodium. Libsodium [8] is a popular library for core cryptographic routines. We chose Minisign [32], a client-only tool for signing files and verifying signatures, as a representative application that uses Libsodium. The private key used for file signing is stored in an object of type `SeckeyStruct`. We annotated the `SeckeyStruct` pointer in `minisign.c` as sensitive. Using the hardened binary, we performed two operations. We first signed a 1GB file using a pre-generated private key, and then verified the signature against the file. Our results of measuring the completion time show that for signing the runtime overhead is 68% (field-ins.) and 33% (field-sen.), while for verification the overhead is 57% (field-ins.) and 35% (field-sen.).

It is important to note that although the verification process does not use the sensitive private key, it still suffers from some performance overhead due to the imprecision of our sensitive data domain construction, in both approaches. This imprecision causes the arguments to the `crypt_hash_sha512`, `crypt_hash_sha512_update`, and `crypt_hash_sha512_final` functions, which compute the hash of the file contents, to be marked as sensitive. As these functions are shared by both signing and verification operations, both operations exhibit a performance overhead.

6.2.6 Results Summary. Our results are summarized in Tables 1 and 2. We observe that for all five applications, only a fraction of all memory accesses had to be instrumented, and as expected, this fraction is lower for the field-sensitive approach. The time taken for the pointer analysis and the value flow analysis (not shown in the table) for the five applications ranges from 20 seconds (for Memcached) to 3 minutes 45 seconds (for Lighttpd).

The performance overhead observed in all five applications varies significantly. The variance is clearly tied to the nature of these applications. For instance, in the MbedTLS server case, the bulk of instrumentation involves only the SSL handshake phase. Data transfer incurs little overhead, and network I/O incurs no overhead. In the `ssh-agent` case, the instrumentation affects only the fetching of the decrypted private key. The rest of the SSH login and network I/O proceeds unchanged. In the Lighttpd case, the instrumentation affects each access, but the sensitive password buffer is accessed sequentially, leading to amortization of the data transformation cost over multiple accesses to the password buffer. On the other extreme, every iteration of the core loop in Minisign that computes the signature of the file is instrumented. Since all operations are local, there is no expensive network I/O, and so the overhead is significantly higher.

Although a direct comparison is not possible due to the different hardware experimental setups, we report significantly lower overhead than solutions based on memory safety. For example, DataShield [27] performs a coarse-grained bounds check on *all* memory accesses, with a more fine-grained bounds check for pointers potentially accessing sensitive data, whereas our solution requires instrumenting only the required sensitive pointers. DataShield reports a higher performance overhead of 35% for the same MbedTLS server application, compared to 13% for our approach. We could not successfully compile the other applications in our test suite with DataShield. Similarly, SoftBound [64], which applies full memory safety, incurs a 116% overhead for the SPEC benchmarks [2]. Moreover, as described in Section 6.3, our solution provides protection against cold boot attacks, as well as side-channel attacks such as Spectre [44], because the sensitive data is present in memory *only* in an encrypted form, unlike in the case of approaches based on memory safety, which only protect pointers. Additionally, the performance overhead of our approach is comparable to the reported overhead of official mitigations for some Spectre attack variants [49].

6.3 Security Evaluation

As a sanity check, we verified that sensitive data is never present unencrypted in main memory. To that end, we used a custom program to repeatedly scan the memory of the running process every two seconds. The program uses the `gcore` tool to attach to the process and dump its memory contents. At the end of the experiment, we scan these memory dumps for the first and last four bytes of the protected data. We verified that for all five applications the sensitive data was not present in an unencrypted form in memory.

Defending against Spectre Attacks. We use a publicly available proof-of-concept to illustrate the effectiveness of our system against Spectre attacks. Figure 3 shows a simplified snippet of the vulnerable code used. The attack begins by passing a chosen value `x`, so that `array1[x]` point to a victim address that the attacker chooses to disclose—in this case, the variable named `secret`. The vulnerability causes `array1[x]` to be loaded, and used to compute the offset into

```
SENSITIVE char *secret = "The␣Secret";
void victimFunction() {
    ...
    if (x < array1_size)
        y = array2[array1[x] * 4096];
    ...
}
```

Figure 3: Simplified example of code vulnerable to the Spectre attack used for our evaluation.

array2, even if the branch condition fails, that is, if x is greater than array1_size. This results in the contents of secret to be loaded into the cache, from where they can be leaked through side channel attacks.

To protect the contents of secret, we annotate it as SENSITIVE. At runtime, its contents are stored only in its encrypted form in memory, and thus also in the hardware caches. As expected, we verified that leaking the contents of the cache via the Spectre attack only returns the encrypted values of the secret variable.

7 LIMITATIONS

In our approach, all loads and stores to variables annotated as sensitive are protected through encryption. Hence, without knowing the secret key, attackers cannot write any desired values to sensitive variables in their correct encrypted form. However, encryption alone does not provide complete protection against attackers who have the capability of performing arbitrary memory writes.

For instance, consider a sensitive variable is_admin related to some authentication operation. Such variables are often checked as part of the program logic by comparing against "not-zero" (e.g., is_admin != 0). In such scenarios, even if the variable is encrypted, attackers can overwrite it with an arbitrary value, and achieve a very high probability of the decrypted value being non-zero. A possible way to address this limitation is to use a message authentication code (MAC) for authenticating writes to sensitive variables, in order to guarantee that only authorized instructions can modify sensitive values. However, it is difficult to identify authorized instructions, especially in case of complex data-only attacks. We leave the exploration of more effective techniques for ensuring data integrity as part of future work.

The dearth of efficient pointer analysis techniques directly impacts the precision of our approach, and its applicability to larger and more complex applications. Ideally, one would want to analyze and transform all libraries that are used by the target application. However, the analysis time depends on the size of the input source code. In our current prototype, to keep the analysis time manageable (i.e., in the order of minutes instead of multiple hours), we excluded Libc from our static analysis passes in order to limit the size of the input source code. Thus, when sensitive arguments are passed to a Libc function, we must first decrypt them.

Nevertheless, to minimize the exposure of decrypted sensitive data to external functions, we turned to custom implementations of commonly used Libc functions, such as memcpy, memcmp, strcpy, and strlen. An immediate direction for future work is to explore other pointer analysis techniques besides Andersen's algorithm (which has a complexity of $O(n^3)$). One possibility is the more efficient unification-based Steensgaard's algorithm [79]. Unlike Andersen's algorithm, however, there is no available implementation (to the best of our knowledge) of Steensgaard's algorithm that could be easily incorporated into the SVF suite [80] or LLVM itself.[1]

It is prudent to note that precise and scalable pointer analysis is an open problem, and other state-of-the-art memory isolation [45] and control flow integrity [83] mechanisms have made similar compromises by opting for overly conservative pointer analysis. We use the best available techniques in a conservative way to avoid false positive issues. We demonstrate that despite incurring a much higher performance penalty than what would be possible with more accurate pointer analysis, our approach still incurs a reasonable performance overhead.

Lastly, because we do not implement runtime key rotation, one can envision a scenario where an adversary can use a known plaintext attack against the sensitive data. However, the data we are trying to protect (i.e., private keys, session cookies) has sufficient entropy to ensure that finding exact matches with 128 bits of known plaintext is hard. Therefore, it is safe to use deterministic encryption under the assumed threat model and goals. Moreover, for this type of attack to be successful, the attacker would require access to an oracle, which falls outside of our threat model.

8 RELATED WORK

8.1 Memory Safety

Enforcing full memory safety to unsafe languages can, in theory, block most memory corruption exploits. In practice, however, the low-level nature of the C and C++ languages, which allow unchecked array indexing, conflation of pointers and arrays, pointer arithmetic, and type casting, makes retrofitting memory safety protections into existing programs a daunting task [63]. The overall strategy for enforcing whole-program memory safety is to maintain bounds information either for each pointer [43, 64–66] or object [13, 50, 75], and to check every pointer dereference against the bounds associated with the target pointer or object. By trading extra memory space for performance, baggy bounds checking [13] is currently one of the most efficient object-based bounds checking approaches, although its performance overhead is still prohibitively high, at an average of 60% for the SPEC benchmarks.

That said, spatial safety in the form of bounds checking alone still cannot prevent use-after-free and double-free vulnerabilities. Approaches that combine both spatial and temporal safety achieve better memory safety, but at an even higher cost. As a case in point, when CETS [65] is coupled with SoftBound [64] to achieve full memory safety, the composition results in an average overhead of 116% for the SPEC benchmarks [65].

Other approaches, such as Diehard [23], Dieharder [67], Cling [11], Archipelago [57], FreeSentry [89], WIT [12], CPI [47], and the works of Dhurjati et al. [33] and Byoungyoung et al. [51], opt for providing weaker guarantees to achieve better performance and compatibility, and thus do not offer complete protection. An alternative trade-off

[1]Although the LLVM compiler toolchain provides a CFL unification-based alias analysis pass named CFL-Steens [5], because the pass performs alias analysis, it must be invoked separately for each pair of memory operands. It then performs a graph search on each query to resolve whether the two operands alias, instead of computing the full points-to graph at once, like SVF. Due to these fundamental differences in the functionality of SVF and CFL-Steens, we leave porting CFL-Steens to SVF as future work.

is made by DataShield [27], which opts to provide full memory safety on only a subset of sensitive data annotated by developers. Although promising, even for an I/O-heavy application such as a TLS server, DataShield still incurs a considerable runtime overhead of 35.7%. Selective data encryption provides a complementary approach, but at a much lower cost.

8.2 Transformation of In-Memory Data

An alternative approach to memory safety is to apply a transformation to the data in the main memory. As long as the attacker can not reverse this transformation, the original data can not be recovered or modified, thus preserving confidentiality and integrity. Data space randomization [20, 24] applies this principle to prevent buffer overflow attacks, using a XOR operation to randomize the in-memory representation of objects. Our work is inspired by this approach to selectively transform sensitive data in memory, but using stronger AES encryption instead.

Memory encryption using AES as a protection against cold boot attacks was proposed by Papadopoulos et al. [68]. While their approach uses a similar decryption cache scheme as ours, we integrate a more robust pointer and value flow analysis to ensure that accesses to sensitive data is always transformed correctly.

8.3 Data Flow Integrity

Similar to control flow integrity techniques, that protect against control flow attacks, data flow integrity mechanisms can protect against data-only attacks. Data Flow Integrity [28] precomputes a valid data flow graph and, at runtime, validates all data flows against it. However, this approach has a significant overhead of 104% for the CPU-bound SPEC benchmarks. Recently, DFI-assisting hardware extensions [78] were proposed to lower the runtime overhead.

8.4 Hardware Based Mechanisms

Hardware-based defenses such as TRESOR [62], PRIME [36], and PixelVault [85] protect sensitive computation from an adversary with physical access to the device. TRESOR and PRIME provide a memory-less, CPU bound infrastructure for sensitive computation, such as RSA encryption. Ginseng [90] protects against an untrusted operating system, by storing sensitive stack variables, strictly in registers, and relies on a secure implementation of *secure stack*, and CFI, in ARM TrustZone's Trusted Execution Environment (TEE) [16]. Likewise, Intel's Software Guard Extensions (SGX) [42] provides a set of CPU instructions that can be used by user mode applications to create private regions, called "enclaves," for sensitive code and data. Various approaches have leveraged this (e.g., [17], [25], [76], [82], [53]), but each involves major restructuring of the source code, including changes to the compiler, OS support, and runtime libraries. The same is true for TRESOR, PRIME, and PixelVault.

MemSentry [45] is a memory isolation framework that allows users to create isolated memory regions by leveraging hardware features. SP^3 [88] and SeCage [55] use hypervisor support to isolate sensitive data on a per-page basis. Compared to these systems, we support a finer-grained separation between sensitive and non-sensitive data, *at the granularity of individual variables.*

9 CONCLUSION

We presented a compiler-level defense that provides strong protection against the emerging threat of data leakage attacks. Our approach allows developers to conveniently annotate program variables or data inputs as sensitive, and ensures that all sensitive data is always kept encrypted when stored in memory.

Unlike existing memory safety or isolation approaches, our solution is geared toward protecting only a subset of a process' data—a design decision that allows for a radically different memory access instrumentation strategy. Instead of instrumenting all memory accesses in the most lightweight manner possible, our solution instruments only a fraction of all memory accesses, and thus enables the use of more heavyweight encryption using AES. Our prototype implementation aptly demonstrates the benefits of the proposed approach, and also highlights important challenges in the area of whole-program fine-grained pointer analysis that, once resolved, will allow faster analysis of more complex applications, and will enable protection against the full spectrum of data-only attacks, by offering data integrity in addition to data confidentiality.

ACKNOWLEDGMENTS

We would like to thank R. Sekar and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through awards CNS-1749895 and CNS-1617902, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, or DARPA.

REFERENCES

[1] 2003. Grsecurity. https://grsecurity.net/.
[2] 2006. SPEC CPU2006 Benchmark. http://www.spec.org/cpu2006.
[3] 2014. The Heartbleed Bug. http://heartbleed.com/.
[4] 2015. Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.
[5] 2016. CFL-based Unification-based Alias Analysis. https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/CFLSteensAliasAnalysis.cpp.
[6] 2016. Intel Streaming SIMD Extensions Technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.
[7] 2018. SSL Library mbed TLS. https://tls.mbed.org/.
[8] 2018. The Sodium crypto library (libsodium). https://www.gitbook.com/book/jedisct1/libsodium/details.
[9] 2019. Overview: Intrinsics for IntelÂő Advanced Vector Extensions 2. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-overview-intrinsics-for-intel-advanced-vector-extensions-2-intel-avx2-instructions.
[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*. 340–353.
[11] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium.*
[12] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*. 263–277.
[13] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *Proceedings of the 18th USENIX Security Symposium.* 51–66.
[14] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.
[15] Brad Antoniewicz. 2013. Analysis of a Malware ROP Chain. http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html.
[16] ARM. 2010. ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI).* 689–703.

[18] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *Proceedings of the International Static Analysis Symposium (SAS).* 84–104.

[19] Andrew Baumann. 2017. Hardware is the new Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS).*

[20] Brian Belleville, Joseph Michael Nash, Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz, Hyungon Moon, Jangseop Shin, Dongil Hwang, Seonhwa Jung, and Yunheung Paek. 2018. Hardware Assisted Randomization of Data. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*

[21] Eli Bendersky. 2015. Pure-python library for parsing ELF and DWARF. https://github.com/eliben/pyelftools.

[22] James Bennett, Yichong Lin, and Thoufique Haq. 2013. The Number of the Beast. http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html.

[23] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 158–168.

[24] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 1–22.

[25] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: confidential ZooKeeper using intel SGX. In *Proceedings of the 17th International Middleware Conference.* 14.

[26] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium.*

[27] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 193–204.

[28] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI).* 147–160.

[29] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium.*

[30] Joan Daemen and Vincent Rijmen. 2013. *The design of Rijndael: AES-the advanced encryption standard.* Springer Science & Business Media.

[31] Jamie Danielson. 2017. ProcessMitigations 1.0.7. https://www.powershellgallery.com/packages/ProcessMitigations/1.0.7.

[32] Frank Denis. 2018. Minisign: A dead simple tool to sign files and verify digital signatures. https://github.com/jedisct1/minisign.

[33] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2003. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES).* 69–80.

[34] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. (XFI): Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI).*

[35] Francisco Falcon. 2015. Exploiting Adobe Flash Player in the era of Control Flow Guard. In *Black Hat Europe.*

[36] Behrad Garmany and Tilo Müller. 2013. PRIME: private RSA infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC).* 149–158.

[37] Dan Goodin. 2007. Safari zero-day exploit nets $10,000 prize. https://www.theregister.co.uk/2007/04/20/pwn-2-own$_w$inner/.

[38] Brian Gorenc. 2017. Pwn2Own 2017 – An Event for the Ages. http://blog.trendmicro.com/pwn2own-2017-event-ages/.

[39] Shay Gueron. 2010. Intel® Advanced Encryption Standard (AES) New Instructions Set. *Intel Corporation* (2010).

[40] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium.* 177–192.

[41] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P).*

[42] Intel. 2015. Software Guard Extensions SDK. https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf.

[43] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference (ATC).* 275–288.

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P).*

[45] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys).* 437–452.

[46] Vadim Kotov. 2014. Dissecting the newest IE10 0-day exploit (CVE-2014-0322). http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[47] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI).* 147–163.

[48] MWR Labs. 2013. MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit. https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/.

[49] Michael Larabel. 2016. The Current Spectre / Meltdown Mitigation Overhead Benchmarks On Linux 5.0. https://www.phoronix.com/scan.php?page=article&item=linux50-spectre-meltdown.

[50] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 129–142.

[51] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, (NDSS).*

[52] Haifei Li. 2011. Understanding and Exploiting Flash ActionScript Vulnerabilities. CanSecWest.

[53] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC).*

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium.*

[55] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS).* 1607–1619.

[56] Chris Lomont. 2016. Introduction to Intel Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[57] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 115–124.

[58] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS).* 941–951.

[59] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space.

[60] Daniel Moghimi. 2014. Subverting without EIP. http://www.moghimi.org/subverting-without-eip/.

[61] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Z. Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2018. Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-oriented Attacks. In *Proceedings of the 3rd IEEE European Symposium on Security & Privacy (Euro S&P).*

[62] Tilo Müller, Felix C. Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Security Symposium.*

[63] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *Proceedings of the 1st Summit on Advances in Programming Languages, (SNAPL).* 190–208.

[64] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 245–258.

[65] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the International Symposium on Memory Management (ISMM).* 31–40.

[66] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526.
[67] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS).* 573–584.
[68] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. 2017. No Sugar but all the Taste! Memory Encryption without Architectural Support. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).* 362–380.
[69] Andrew Pardoe. 2017. Security Features in Microsoft Visual C++. https://blogs.msdn.com/vcblog/security-features-in-microsoft-visual-c/.
[70] PaX Team. 2003. Address Space Layout Randomization. http://pax.grsecurity.net/docs/aslr.txt.
[71] PaX Team. 2003. Paging Based Non-executable Pages. http://pax.grsecurity.net/docs/pageexec.txt.
[72] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4.
[73] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium.*
[74] Roman Rogowski, Micah Morton, Forrest Li, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2017. Revisiting Browser Security in the Modern Era: New Data-only Attacks and Defenses. In *Proceedings of the 2nd IEEE European Symposium on Security & Privacy (Euro S&P).*
[75] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS).* 159–169.
[76] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P).* IEEE, 38–54.
[77] Fermin J. Serna. 2012. CVE-2012-0769, the case of the perfect info leak. http://zhodiac.hispahack.com/my-stuff/security/Flash$_A$SLR$_b$ypass.pdf.
[78] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*
[79] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL).* 32–41.
[80] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC).* 265–266.
[81] Bing Sun, Chong Xu, and Stanley Zhu. 2017. The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation. In *Black Hat Asia.*
[82] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC).*
[83] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS).* 927–940.
[84] Stephan van Schaik, Alyssa Milburn, Sebastian ÃŰsterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P).*
[85] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS).* 1131–1142.
[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP).* 203–216.
[87] David Weston and Matt Miller. 2016. Windows 10 Mitigation Improvements. Black Hat USA.
[88] Jisoo Yang and Kang G. Shin. 2008. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *ACM International Conference on Virtual Execution Environments (VEE).* 71–80.
[89] Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, (NDSS).*
[90] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*
[91] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS).*

A APPENDIX: ADDITIONAL IMPLEMENTATION DETAILS

In Section 5 we discussed the encryption transformation which we apply to the merged intermediate representation code. During our experiments with applying our approach to various applications, we encountered several corner cases that required special consideration, when applying this transformation. In what follows, we discuss the most important ones and how we addressed them.

A.1 Object Alignment

Although AES operates on 128-bit data blocks, sensitive objects come in various smaller and larger sizes. To accommodate any object size, we round up the size of sensitive objects to multiples of 128 bits, and allocate them strictly on 128-bit boundaries. We handle global, stack, and heap objects in the following ways.

Global and Stack Variables: LLVM's IR supports the specification of alignment for global and stack variables. We use this feature to specify a custom alignment of 16 bytes for sensitive global and stack variables and round up the size of these variables to a multiple of 128 bits.

Heap Variables: We must ensure that the alignment requirements are respected for objects allocated dynamically on the heap. To achieve this, we provide custom memory allocation functions. These custom memory allocation functions use the `posix_memalign` function to allocate memory aligned to 128 bit boundaries. We also round up the size of the allocated region to the nearest multiple of 128 bits. Then, as part of our memory encryption transformation, all sensitive calls to memory allocation library functions, such as `malloc` and `calloc`, are automatically replaced with our custom memory allocation functions.

A.2 Globals with Default Initializers

When global variables are initialized to default values, their memory is allocated in the `.data` segment and is initialized to the specified value at compilation time—there are no explicit `StoreInst` instruction executed at runtime. Because our AES instrumentation transforms explicit memory loads and stores, we must handle the initialization of global variables in a separate way. This is achieved by introducing an `encrypt_globals` function that encrypts all sensitive global variables, and inserting a call to this function at the start of the `main` function.

A.3 Sensitive Constants

The sensitive data domain may include constants which must be encrypted in memory. By default, LLVM allocates constants in the `.rodata` section, which is a read-only section. Attempting to write to these objects as part of the encryption process would cause a protection fault. In our implementation, we address this problem by removing the `constant` specifier for these objects.

A.4 Environment variables

The sensitive data domain may include pointers to environment variables, such as `$HOME`, which can end up being marked as sensitive as a result of the over-approximation of our pointer analysis.

```
Data: List of objects annotated as SENSITIVE, sensitive value
      flow sinks, pts-to and pts-from information
Result: List of all objects in the sensitive equivalence class
eq_class := List of objects annotated as sensitive;
new_objs := eq_class;
while !new_objs.isEmpty() do
    ptr_set := ptrs which can point to objs in eq_class;
    ptr_targets := targets of all ptrs in ptr_set;
    new_objs := ptr_targets \ eq_class ;
    eq_class := eq_class ∪ ptr_targets;
end
```

Algorithm 1: Find sensitive data equivalence class.

Encrypting these environment variables causes system calls such as `fopen` to break, and thus these variables must not be modified. We provide a `cloneenv` function, that first clones the value of the environment variable, and returns a pointer to the cloned version. Our memory encryption transformation replaces all sensitive calls to the libc function `getenv` with this `cloneenv` function.