EnGarde: Mutually-Trusted Inspection of SGX Enclaves

Hai Nguyen and Vinod Ganapathy
Department of Computer Science, Rutgers University
{hdn11, vinodg}@cs.rutgers.edu

Abstract— **Intel's SGX architecture allows cloud clients to create enclaves, whose contents are cryptographically protected by the hardware even from the cloud provider. While this feature protects the confidentiality and integrity of the client's enclave content, it also means that enclave content is completely opaque to the cloud provider. Thus, the cloud provider is unable to enforce policy compliance on enclaves.**

In this paper, we introduce EnGarde, a system that allows cloud providers to ensure SLA compliance on enclave content. In EnGarde, cloud providers and clients mutually agree upon a set of policies that the client's enclave content must satisfy. EnGarde executes when the client provisions the enclave, ensuring that only policy-compliant content is loaded into the enclave. EnGarde is able to achieve its goals without compromising the security guarantees offered by the SGX, and imposes no runtime overhead on the execution of enclave code. We have demonstrate the utility of EnGarde by using it to enforce a variety of security policies on enclave content.

1. Introduction

In recent years, the research community has devoted much attention to security and privacy threats that arise in public cloud computing platforms, such as Amazon EC2 and Microsoft Azure. From the perspective of cloud clients, one of the chief security concerns is that the computing infrastructure is not under the client's control. While relinquishing control frees the client from having to procure and manage computing infrastructure, it also exposes the client's code and data to cloud providers and administrators. Malicious cloud administrators can compromise client confidentiality by reading sensitive code and data directly from memory images of the client's virtual machines (VM). They could also inject malicious code into client VMs, *e.g.,* to insert backdoors or log keystrokes, thereby compromising integrity. Even otherwise honest cloud providers could be forced to violate client trust because of subpoenas.

Intel's SGX architecture [7, 15, 17, 18, 22] offers hardware support to alleviate such client security and privacy concerns. SGX allows client processes and VMs to create *enclaves*, within which they can store and compute on sensitive data. Enclaves are encrypted at the hardware level using hardware-managed keys. SGX guarantees that enclave content that includes enclave code and data is not visible in the clear outside the enclave, even to the most privileged software layer running on the system, *i.e.,* the operating system (OS) or the hypervisor. SGX also offers support for enclave attestation, thereby providing assurances rooted in hardware that an enclave was created and bootstrapped securely, without interference from the cloud provider. With SGX, clients can therefore protect the confidentiality and integrity of their code and data even from a malicious cloud provider or administrator, so long as they are willing to trust the hardware.

Despite these benefits, SGX has the unfortunate consequence of flipping the trust model that is prevalent on contemporary cloud platforms. On non-SGX platforms, a benign cloud provider benefits from the ability to inspect client code and data. The cloud provider can provide clients with services such as malware detection, vulnerability scanning, and memory deduplication. Such services are also beneficial to benign clients. The cloud provider can check client VMs for service-level agreement (SLA) compliance, thereby catching malicious clients who may misuse the cloud platform in various ways, *e.g.,* by using it to host a botnet command and control server. In contrast, on an SGX platform, the cloud provider can no longer inspect the content of a client's enclaves. This affects benign clients, who can no longer avail of cloud-based services for their enclaves. It also benefits malicious clients by giving them free reign to perform a variety of SLA-violating activities within enclaves. Researchers have discussed the possibility of such "detection-proof" SGX malware [10, 28, 29]. Without the ability to inspect the client's code, the cloud provider is left to using other, indirect means to infer the presence of such malicious activities. For example, minibox [20] verifies that an application behaves properly by checking system call parameters of the application for malicious activities such as accessing sensitive files that do not belong to the application.

Can a benign client benefit from the security offered by the SGX while still allowing the cloud provider to exert some control over the content of the client's enclaves? One strawman solution to achieve this goal is to use a trusted-third party (TTP). Both the cloud provider and client would agree upon a certain set of policies/constraints that the enclave content must satisfy (as is done in SLAs). For example, the cloud provider could specify that the enclave code must be certified as clean by a certain anti-malware tool, or that the enclave code be produced by a compiler that inserts security checks, *e.g.,* to enforce control-flow integrity or check for other memory access violations. They inform the TTP about these policies, following which the client

discloses its sensitive content to the TTP, which checks for policy compliance. The cloud provider then allows the client to provision the enclave with this content.

However, the main drawback of this strawman solution is the need for a TTP. Finding such a TTP that is acceptable to both the cloud provider and the client is challenging in real-world settings, thereby hampering deployability. TTPs could themselves be subject to government subpoenas that force them to hand over the client's sensitive content. From the client's perspective, this solution provides no more security than handing over sensitive content to the cloud provider.

Contributions. In this paper, we present EnGarde, an enclave inspection library that achieves the above goal without TTPs. Cloud providers and clients agree upon the policies that enclave code must satisfy and encode it in EnGarde. Thus, cloud providers and clients mutually trust EnGarde with policy enforcement. The cloud provider creates a fresh enclave provisioned with EnGarde, and proves to the client, using SGX's hardware attestation, that the enclave was created securely. The client then hands its sensitive content to EnGarde over an encrypted channel. It provisions the enclave with the client's content only if the content is policy-compliant. If not, it informs the cloud provider, who can prevent the client from creating the enclave.

EnGarde's approach combines the security benefits of non-SGX and SGX platforms. From the cloud provider's perspective, it is able to check client computations for policy-compliance, as in non-SGX platforms. From the client's perspective, its sensitive content is not revealed to the cloud provider, preserving the security guarantee as offered by SGX platforms. Moreover, EnGarde statically inspects the client's enclave content only once—when the enclave is first provisioned with that content. One can also imagine an extension of EnGarde that instruments client code to enforce policies at runtime, but our current implementation only implements support for static code inspection. Thus, except for a small increase in enclave-provisioning time, EnGarde does not impose any runtime performance penalty on the client's enclave computations. We have implemented a prototype of EnGarde and have used it to check a variety of security policies on a number of popular open-source programs running within enclaves.

2. SGX Background

Enclaves. The main feature of SGX is its support for enclaves. An enclave is a linear span of a process's virtual address space whose physical pages are drawn from a region of physical memory called the encrypted page cache (EPC). The contents of EPC pages are protected cryptographically by the hardware, which does not reveal the encryption key even to the most privileged software layer on the system (*e.g.,* the OS or the hypervisor). A process can have multiple enclaves in its address space.

A process enters an enclave via an instruction (`EENTER`). Within an enclave, the process can have multiple threads of execution. Each such thread can freely access the memory contents of both the enclave as well as the rest of the process address space. If an enclave thread references an address within the enclave, the hardware fetches corresponding memory page from the EPC and decrypts it within the hardware cache hierarchy, thereby offering the process a view of the plaintext content of the page. An adversary outside the enclave (*e.g.,* observing the memory bus) will only see encrypted traffic to the EPC page, thereby preserving the confidentiality and integrity of the EPC page. SGX imposes a few restrictions on the code that can execute within an enclave. An enclave can only execute user-mode code and cannot invoke any OS services, *e.g.,* via system calls. If the enclave code needs to avail of such services, it must save the enclave state, exit the enclave (via an instruction called `EEXIT`), and have the non-enclave code of the process access such services on its behalf. SGX offers various data structures to save enclave state in an encrypted fashion, thereby protecting it from adversaries outside the enclave. SGX hardware ensures that code executing outside the enclave, whether in user-mode context in the process address space or in kernel-mode context within the OS (or hypervisor), cannot access the plaintext enclave content.

Although an OS (or hypervisor) cannot view the plaintext contents of a process's enclaves, it is still responsible for various aspects of enclave management. The OS creates enclaves for processes (using `ECREATE`), adds or removes pages from a process's enclaves (using `EADD` and `EREMOVE`, respectively), and manages the process's page tables. Page table entries corresponding to the virtual address range of an enclave will be mapped to the EPC. Although we have only introduced a handful of instructions, the SGX supports a total of 24 new enclave management instructions [17, 18].

Attesting and Provisioning Enclaves. When an enclave is newly created within a process's address space, it is initialized with some generic bootstrap code. The exact nature of this bootstrap code differs based on the software vendor who offers this code. However, at the very minimum, this bootstrap code implements basic cryptographic functionality (*e.g.,* for SSL/TLS), wrappers for system calls and other popular libraries that the client's enclave code may wish to use. SGX offers support for *attestation* [7], which allows remote clients of an SGX-based cloud platform to ensure that enclaves are initialized securely.

Remote attestation on SGX platforms follows a standard challenge/response scheme as in TPM-based attestation protocols [30]. The client sends a challenge to the SGX-based machine on the cloud platform. Each SGX-based machine is endowed with a dedicated, Intel-provided *quoting enclave*. The quoting enclave obtains a measurement (a SHA-256 digest of a log of all activities during enclave initialization [7], obtained via the `EREPORT` instruction) of each newly-created enclave, and signs the measurement using a device-specific private key, called the Intel EPID key. The SGX hardware ensures that only the quoting enclave has access to the EPID key. The client can then verify the signed measurements, thereby obtaining a guarantee, rooted in SGX hardware, that the enclave was initialized correctly.

Following attestation, the client *provisions* the enclave

with sensitive content. Thus, the client needs an encrypted, authenticated channel between its server and the newly-created enclave on the cloud platform. On SGX systems, this problem is addressed by generating an ephemeral public/private key pair during enclave creation and initialization. The value of this ephemeral public key is included in the attestation quote that is signed by the quoting enclave, thereby providing the client a hardware-rooted guarantee that binds the public-key to the enclave. The client can then use this public-key to bootstrap an SSL/TLS handshake, thereby establishing a secure channel to the enclave. The client then transmits all sensitive content to the enclave over this encrypted channel.

3. Design of EnGarde

Problem Definition. Given the features of the SGX, a client's enclave is opaque to the cloud provider. This benefits clients because it protects the confidentiality and integrity of their sensitive content. However, the cloud provider can no longer inspect or enforce any policies on enclave content.

In this paper, we remedy the situation by introducing EnGarde, which statically checks the policy compliance of the code that the client proposes to execute in its enclaves. The client and cloud provider agree upon a set of policies that the client's code must satisfy. For instance, the cloud provider may require the client to instrument its code with certain security checks or link its code against certain versions of libraries. EnGarde's architecture supports plugging in *policy modules*, which check compliance based upon the policies that the cloud provider and client mutually agree upon. EnGarde executes during enclave provisioning, and checks that the client's enclave code is policy-compliant. If the client's code is not policy compliant, EnGarde informs the cloud provider, who can prevent code from executing.

Threat Model. We assume that the cloud provider and client are mutually distrusting. Before allowing the client to create and provision enclaves, the cloud provider and client mutually agree upon the set of policies that the client's code must satisfy. We assume that the code of EnGarde and its policy modules is available to both the cloud provider and client for inspection.

From the cloud provider's perspective, the client will attempt to violate the mutually agreed-upon policies. It therefore verifies that EnGarde and its policy modules indeed enforce these policies. From the client's perspective, the cloud provider will attempt to learn the contents of the enclave. Thus, the client verifies that EnGarde and its policy modules leak no additional information about its code to the cloud provider, *i.e.,* the only explicit communication between EnGarde and the cloud provider must be to inform the cloud provider about policy compliance and to identify the virtual addresses of the pages that contain the client's code. For this paper, we do not consider implicit and covert communication channels via which EnGarde can communicate information about the client's code to the cloud provider; techniques to analyze EnGarde's code for such covert channels can be the subject of future research. The client can also use EnGarde to independently verify policy compliance of the enclave code that it wants to provision.

Both the cloud provider and the client trust the SGX hardware platform. EnGarde and its policy modules are loaded into a freshly-created enclave (as part of the bootstrap code). Both the provider and the client use SGX's attestation features to ensure that EnGarde was correctly loaded into the enclave. EnGarde receives the client code over a SSL/TLS channel, checks policy compliance, and informs the cloud provider. Any attempts by the cloud provider to cheat, *e.g.,* by falsely claiming that the code is not policy-compliant or failing to allow policy-compliant code to execute, can easily be detected by the client.

Overall Design. EnGarde primarily consists of in-enclave components that are loaded when an enclave is created (see Figure 1). As is standard on all SGX systems, the client first ensures (using SGX's attestation protocols [7]) that the enclave was initialized securely.

Following this step, the client sets up an end-to-end encrypted channel with the enclave. To do this, the bootstrap code loaded into a freshly-created enclave first generates a 2048-bit RSA key pair and then establishes a socket connection to the client machine. As a next step, the enclave sends its newly-generated public key to the client machine so that it can encrypt its 256-bit AES key and sends the encrypted AES key back to the loader. This key is used to establish an end-to-end encrypted channel with the client.

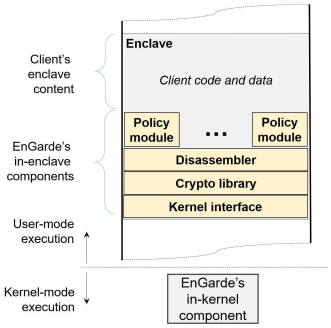

Figure 1. Design of EnGarde.

EnGarde checks the client's enclave contents for policy compliance after the client sends it the contents, but before the content is provisioned within the enclave for execution. The client sends the content in encrypted blocks, which EnGarde's crypto library decrypts to form an in-memory executable representation.

EnGarde operates at the granularity of memory pages, and therefore splits the content into page-level chunks. We assume that the client sends x86 binary code and identifies pages which contain code. The remaining pages are assumed to contain data. EnGarde rejects pages that contain mixed code and data. We assume that clients suitably compile their code so as to satisfy this assumption. We also assume that the enclave code is not obfuscated to hinder analysis.

Once EnGarde has received all the code, it proceeds to disassemble the client's code. To do this, EnGarde relies on the disassembler provided by Google's Native Client (NaCl) [42]. NaCl makes a number of assumptions to ensure clean, unambiguous disassembly. For example, it requires no instructions to overlap a 32-byte boundary, that all control-

transfers target valid instructions, and that all valid instructions are reachable from the start address. EnGarde requires the client's enclave to satisfy the same constraints.

After disassembling the code, EnGarde checks the code for policy compliance. Recall that the specific policies that EnGarde checks depend on those negotiated with the client. In general, the policies can check structural properties of the code, *e.g.,* that certain instrumentation has been added to the code. EnGarde checks policies using pluggable *policy modules*. Each policy module checks compliance for a specific property, and specific policy modules that are loaded during enclave creation depend upon the policies that the client and cloud provider have agreed upon. In Section 5, we discuss three examples of policy modules that we have implemented in our current EnGarde prototype. While EnGarde's disassembler works even on stripped binary code (*i.e.,* code without symbol-table information), specific policy modules may require symbol table information to check compliance. If EnGarde's policy modules require such information, then it requires the client to produce code using symbol tables.

The policy modules determine whether the client's code is policy compliant. If not, the code is rejected, and the enclave is not provisioned. If EnGarde determines that the code is policy-compliant, it then informs the host. EnGarde also contains a host-level component, either running within the host's OS kernel or the hypervisor (if the host is virtualized). EnGarde's in-enclave components provide the in-kernel component with a list of executable code pages. The underlying OS component marks these pages as executable, but not writable. The remaining pages are given write permissions, but are not given execute permissions. The host OS component of EnGarde also prevents the enclave from being extended after it has been provisioned. This ensures that the client cannot inject any further code into the enclave after it has been checked for compliance. EnGarde's in-kernel component enforces execute and write permissions by setting page-table permission bits in the underlying host OS. While the current version of SGX hardware allows for page permissions to be set/cleared by the host OS, it does not yet offer support for page permissions at the hardware level (*i.e.,* page permissions for EPC pages). This feature has been proposed in version 2 of the SGX instruction set [6]. Although EnGarde can be implemented readily even on SGX version 1 processors, the permission check can only be enforced in software within the host OS, and this has been shown to be open to attack [39]. Thus, EnGarde requires the features of SGX version 2 for security.

Following this, the enclave can be accessed and executed as on traditional SGX platforms. Note that EnGarde only operates during enclave provisioning. Thus, EnGarde only imposes a performance penalty during enclave provisioning. Enclave execution incurs no additional runtime overhead.

4. Implementation

Features of the SGX are now commercially deployed in Intel's Skylake series of processors. Despite the availability of commodity hardware, for this paper, we chose to develop EnGarde atop OpenSGX [19], a QEMU-based SGX emulation infrastructure. Two factors governed our choice.

First, open-source software support for SGX enclave development is still rudimentary. To create a system that fully realizes the power of enclaves, we need support for in-enclave bootstrap code and supporting libraries, drivers within the OS, and compiler support to emit SGX code. Although Intel provides SDKs for Windows 10, these SDKs are closed-source, which complicates extension and modification [2]. An open-source Linux SDK [43], which we could have extended, was released only in June 2016 when we were already underway with our EnGarde prototype. While Intel's programming references [17, 18] specify the semantics of instructions, they offer considerable freedom to end-developers to choose their enclave programming model. Community consensus has yet to emerge on these programming models, and rather than define one of our own, we chose to use the programming model defined by OpenSGX. Moreover, OpenSGX incorporates driver support for SGX, and has ported various utilities and libraries to work in enclave mode, which we could readily utilize and extend for EnGarde.

Second, even the SGX architecture itself is evolving. Skylake processors currently implement version 1 of the instruction set. This instruction set poses a number of restrictions [21, 40], the chief of which is that it does not permit page protections to be changed at the hardware level for pages in the EPC. Page protections can still be changed at the level of page tables, and SGX performs a two-level page protection check prior to writing or modifying a page: at the page-table level and at the hardware level. However, recent work has shown that lack of support for page protection modifications at the EPC level can be exploited [39]. As already discussed, EnGarde relies on the ability to change EPC page protections. In addition, SGX hardware currently requires all enclave memory to be committed at enclave build time (therefore requiring the developer to predict and use the maximum stack/heap sizes during enclave build) and does not allow additional code modules to be dynamically loaded into the enclave after it has been built. While these changes have been proposed in version 2 of the instruction set [21, 40], it is not yet commercially available [6]. In contrast, it is easy to explore such changes within the context of a software-based SGX emulator such as OpenSGX.

Our EnGarde prototype supports x86-64 executables that use ELF format [11, 32], are compiled as position independent executables and are statically linked. In this section, we first describe our modifications to OpenSGX. We then discuss the components of EnGarde.

Modifications to OpenSGX. The client enclave holds the client executable as well as its decoded instructions. As a result, the number of EPC pages should be large enough to meet the memory requirements of the client enclave. OpenSGX restricts the number of EPC pages to 2000. We modified OpenSGX to increase the default number of EPC pages to 32000 which translates to 128 MB for the physical protected memory region. On OpenSGX, this size can be

extended to meet further memory requirements. We also change the number of initial page frames for the heap region from the default value of 300 to 5000.

Binary Disassembly. The executable file provided by the client is in 64-bit ELF format. An ELF binary comprises of several segments and each segment has one or more sections. Each section contains information of similar type, for instance the `.text` section contains the executable code, all writable data is stored in the `.data` section and uninitilized data is kept in the `.bss` section. The ELF format also features an ELF header located at the beginning of the file and is used to recognize other parts of the file.

One common challenge in disassembling a binary is mixing of code and data within the code section. Our EnGarde prototype assumes that the client's executable is compiled with separated code and data sections. Before disassembling the code sections of the executable, the loader checks its header to verify that the executable is correctly formatted. The checks include checking the signature as well as the ELF class of the executable. The loader next reads the program header of the executable to extract all text sections. We implement the disassembler based on the 64-bit binary disassembler of NaCl, an open source sandbox for native code. Using prefix and opcode tables for x86-64 bit instruction set, the disassembler parses the byte sequence of the text sections into instructions and associated metadata information, *e.g.,* the number of prefix bytes, number of opcode bytes and number of displacement bytes [5].

NaCl's disassembler does not track all disassembled instructions. Instead, during the disassembly it uses a buffer that stores the most recently disassembled instructions. This stems from the fact that the NaCl validates each instruction right after it is disassembled. We instead use a dynamically allocated buffer that can hold all the instructions and use that buffer as the input to policy checks. Since dynamic memory allocation involves exiting the enclave mode and invoking a trampoline, we reduce the involved overhead by restricting the calls to `malloc` by allocating a memory page at a time instead of just a memory region for an instruction.

Along with disassembling the executable, the loader also reads the symbol tables to keep track of the address and name of all the functions in the executable. It constructs a symbol hash table whose key is the address of a function and value is the name of the function. This symbol hash table could be used by the policy checking component when it perform policy checks.

Loading. After the executable has been checked and confirmed to follow certain policies the loader takes over. The loader maps the `text`, `data` and `bss` segments to the enclave memory, making the `text` segment be executable but read-only, the `data` segment and `bss` segment be writable but non-executable. It then locates the sections that require relocations and the locations at which the relocations should be applied. The loader acquires all the information that it needs for relocations from the `.dynamic` section of the executable. In particular, the loader determines the address and the size of relocation tables which contain detailed information for relocations by reading appropriated entries of the `.dynamic` section. Upon completing relocation, the loader sets up a call stack and transfers control to the executable.

Components	LOC
Code Provisioning	270
Loading and Relocating	188
Checking Executables linked against musl-libc	1,949
Checking Executables Compiled with Stack Protection	109
Checking Executables Containing Indirect Function-Call Checks	129
Client's side program	349
Musl-libc	90,728
Lib crypto (openssl)	287,985
Lib ssl (openssl)	63,566
Total	453,349

Figure 2. Sizes of various components of EnGarde. Some of these components (*e.g.,* the cryptographic libraries) are part of the default loader in all enclave implementations.

5. Evaluation

In evaluating EnGarde, our main goals were to demonstrate the flexibility of EnGarde by showing that it can check compliance against a variety of policies, and understand the performance costs of various components of EnGarde.

Our setup consisted of running OpenSGX atop of Ubuntu 14.04 on a physical machine equipped with an Intel Core i5 CPU and 16GB of memory. We use clang and llvm version-3.6 to compile and instrument many real world applications to run within enclaves: Nginx (an HTTP server), Memcached (a popular key-value store), Netperf (a networking benchmark), otp-gen (a password generator), graph-500 (a graph data benchmark) and two SPEC benchmarks (401.bzip2 and 429.mcf). In all experiments, all the applications are compiled as position independent executables and are statically linked. To keep the size of the executables small all applications are linked against musl-libc [3] instead of GNU libc [1]. Figure 2 shows the lines of code of all the components of EnGarde's implementation. In the following sections, we describe the performance costs of three policy modules that we implemented in EnGarde. For each benchmark, we assume that the benchmark executes within the enclave, and we evaluate the cost of EnGarde as it loads the benchmark within the enclave for execution, and checks for policy compliance.

Compliance for Library Linking. When a cloud provider allows a client to run code on its platform, it often expects the client to run a particular version of the code. For example, the cloud provider may require that the client execute a version of the code that has been patched with the latest security updates. As a special case of this, the cloud provider may wish to check whether the client's code has been linked against specific versions of certain libraries. For example, the cloud provider may wish to ensure that if the client's code uses OpenSSL, then the version of OpenSSL that is used is free of the vulnerability that caused the HeartBleed exploit. As another example, consider `/CONFIDENTIAL` [36], a library that ensures that enclave code satisfies certain information-flow confinement properties, *i.e.,* enclave code that is linked against this library will not accidentally leak

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	262,228	694,405,019	1,307,411,662	128,696
401.bzip2	24,112	34,071,240	148,922,245	4,239
Graph-500	100,411	140,307,017	246,669,796	4,582
429.mcf	12,903	18,242,127	123,895,553	4,363
Memcached	71,437	137,372,517	489,914,732	8,115
Netperf	51,403	90,616,563	367,356,878	18,090
Otp-gen	28,125	42,823,024	198,587,525	5,388

Figure 3. Performance of EnGarde to check the *Library-linking* policy. Here EnGarde checks whether each benchmark has been linked against musl-libc. The figure shows the size of each benchmark, measured as the number of instructions in the code to be loaded in the enclave, and the time taken to execute each step of EnGarde, reported as CPU cycles. Wall-clock time can be obtained by multiplying CPU clock cycles with the clock cycle time. A CPU with a clock rate of 3.5GHz as used in our experiments has 1/3.5 nanoseconds cycle time. Therefore, the 694,405,019 cycles it takes to disassemble Nginx, for example, consumes 198.4 milliseconds.

sensitive information. To prevent liability issues arising from any accidental data leaks in the client's code, the cloud provider may wish to ensure that the client's code is linked with the `/CONFIDENTIAL` library.

To illustrate the power of EnGarde at enforcing such library-linking policies, we implemented a policy module that verifies whether an executable is linked against musl-libc [3] version 1.0.5. To perform this check, we first generate the SHA-256 hashes of all the functions of musl-libc v1.0.5. For enforcement, the policy module iterates through the instruction buffer of the code to be loaded in the enclave, and looks for all direct function calls. For each direct function call, the policy check computes the target of the call and then looks up the symbol hash table to get the function name of the target. If the target does not exist in the symbol hash table the check will mark the function call as invalid; otherwise, it will compute the SHA-256 hash of all the instructions of the function. Specifically, the policy module sequentially reads instructions starting from the computed target address and stops when it comes across an instruction that is at the beginning of another function. The policy module relies on the symbol hash table to identify whether an instruction address is at the beginning of a function. The policy check next compares the hash of the function in the executable with its hash in musl-libc. If the two hashes do not match, the client has not provided the required musl-libc; otherwise, the policy check continues with the next iteration until it reaches the end of the instruction buffer.

To compute the performance cost, we adopt the approach suggested in the OpenSGX paper [19] and assume that each SGX instruction takes 10K CPU cycles and non-SGX instructions run at native speed within the enclave. We leverage OpenSGX's performance counter and QEMU's instruction count [4] to count SGX and non-SGX instructions. We calculate the CPU cycles of non-SGX instructions by measuring the instructions per cycle by executing the loader natively without OpenSGX. Figure 3 presents the results of our experiments when running this policy check against different benchmarks.

Compliance for Stack Protection. Given the prevalence of buffer overflow vulnerabilities in low-level code, a number of modern compilers now give the option of emitting extra code to protect loads and stores to memory locations. Clang's `-fstack-protector` flag lets the LLVM compiler add a guard variable when a function starts and checks the variable when a function exits. For instance, when compiled with the flag, the following extra code is emitted:

```
19311:   mov %fs:0x28, %rax
1931a:   mov %rax, (%rsp)
193fe:   mov %fs:0x28, %rax
19407:   cmp (%rsp), %rax
1940b:   jne 1941f
1941f:   callq 8d5bf <__stack_chk_fail>
```

The two instructions at addresses `193fe` and `19407` check if the variable at the top of the stack is the same as the variable at `%fs:0x28`. If the values do not match, control will be transfered to the `__stack_chk_fail` function.

Clang also provides the `-fstack-protector-all` option which is similar to `-fstack-protector` except that *all* functions are protected. To check whether an executable is compiled with this flag, the policy module iterates through the instruction buffer and identifies the start of a function using the symbol hash table. Within each function, the policy check looks for instructions that affect the stack's variables (*e.g.,* `mov %rax,(%rsp)` in the above example). It then identifies the source operand of the instruction (`%rax`) and figures out the value of the source operand (`mov %fs:0x28,%rax`). As a next step, it checks if the function contains a `cmp` instruction with the source and destination are the stack's variable and the previous source operand, respectively. It also has to check that just preceding the `cmp` instruction, there is an instruction that computes the original value of the source operand (`mov %fs:0x28,%rax`). Finally, the policy looks for the `jne` and `callq` instructions. It computes the target of the `callq` instruction and checks the symbol hash table to verify that the target corresponds to the the `__stack_chk_fail` function.

Of course, our implementation of EnGarde's policy module is customized for Clang's stack protection instrumentation as emitted by the `-fstack-protector` flag. It can easily be customized to check stack protection instrumentation inserted by other tools, such as Google's AddressSanitizer [33], LLVM SoftBound [23], etc. Figure 4 presents the results of our experiments when running this policy check against different benchmarks executing in enclaves.

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	271,106	719,360,640	713,772,098	128,662
401.bzip2	24,226	34,292,136	862,023,613	4,206
Graph-500	100,488	140,588,361	195,218,892	4,548
429.mcf	12,985	18,288,921	31,459,881	4,330
Memcached	71,677	137,877,497	325,442,403	8,081
Netperf	51,868	91,577,335	183,274,713	18,057
Otp-gen	28,217	43,053,386	217,302,816	5,355

Figure 4. Performance of EnGarde to check the *Stack protection* policy. As before, for performance numbers, we report the CPU cycles.

Restricting Indirect Function Calls. Protecting applications against control-flow hijacking attacks is one of the emerging concern due to the fact that attackers have recently focused on taking advantage of heap-based corruptions to overwrite function pointers to change the flow of a program. Control-

Benchmark	#Inst.	Disassembly	Policy Checking	Loading and Relocation
Nginx	267,669	821,734,999	20,843,253	128,668
401.bzip2	24,201	34,235,817	1,751,276	4,206
Graph-500	100,424	140,429,738	7,014,913	4,548
429.mcf	12,903	18,242,127	1,177,429	4,330
Memcached	71,508	138,231,446	5,301,168	8,081
Netperf	51,431	91,161,601	3,775,318	18,057
Otp-gen	28,132	42,829,680	2,334,847	5,355

Figure 5. Performance of EnGarde to check the *Indirect Function-Call* policy. As before, for performance numbers, we report the CPU cycles.

flow Integrity (CFI)is a measure that guards against these attacks by restricting the targets of indirect control transfers to a set of precomputed locations.

We implemented a policy check to verify that executables are compiled with indirect function-call checks as proposed in recent work by Google (IFCC) [37]. IFCC protects indirect calls by generating instrumentation for the targets of indirect calls. It adds code at indirect call sites to ensure that function pointers point to a jump table entry. For example, the LLVM implementation of IFCC emits the following code for an indirect function call:

```
1b459:   lea 0x85c70(%rip), %rax
         #<__llvm_#jump_instr_table_0_1>
1b460:   sub %eax, %ecx
1b462:   and $0x1ff8, %rcx
1b469:   add %rax, %rcx
1b475:   callq *%rcx
```

To instrument executables with these checks, we use the LLVM/clang toolchain enhanced with the IFCC patch [26]. To check whether an executable is compiled with IFCC checks, EnGarde's policy module first figures out the range of the jump table by relying on the fact that all jump table entries have the following format:

```
a19d0 <__llvm_jump_instr_table_0_289>:
a19d0:       jmpq 41090 <ngx_execute_proc>
a19d5:       nopl (%rax)
```

EnGarde's policy module for this check iterates through the instruction buffer and looking for indirect function calls. It then verifies that before the indirect function calls, there is a sequence of instructions `lea`, `sub`, `and` and `add`, with data dependence between registers as shown in the code snippet above. It then computes the target of the indirect call and verifies that the target is within the range of the jump table. Figure 5 presents the results of our experiments when running this policy check against different benchmarks.

6. Related Work

Intel SGX. A number of recent projects have applied Intel SGX for trusted computation on cloud platforms. Microsoft's Haven [8] was the first project that leveraged Intel SGX to enable unmodified Windows binaries to run on Intel SGX-based cloud platforms. Haven allows an application to be linked with a runtime library OS variant of Windows 8 and loaded into an enclave. The confidentiality and integrity of this code and data is protected from the cloud provider. VC3 [31] is another effort to leverage SGX to provide security for enclaves that perform MapReduce-style computations. VC3 also recognized that enclave code with memory safety errors could pose a threat to confidentiality of client data, and proposed instrumenting client code with a form of control-flow integrity instrumentation.

SecureKeeper [9] leverages Intel SGX to keep ZooKeeper-style computations confidential. S-NFV [34] uses Intel SGX to address security issues of today's Net-Work Function Virtualization (NFV) infrastructures by securely move the states of NFV applications in enclaves. SGX processors are also used to improve the performance of privacy preserving multi-party machine learning [25].

While SGX provides attractive hardware-based security guarantees, it places considerable burden on the enclave code programmer to ensure that computations executing within the enclave do not accidentally leak information. Similarly, vulnerabilities such as memory safety errors in enclave code can lead to exploits that leak confidential data. Moat [14] takes a first step towards this goal by statically analyzing x86 machine code to be loaded within the enclave to check for information-flow violations. `/CONFIDENTIAL` [36] extends the approach proposed in Moat, and provides enclave authors with a library that they can link their enclave code against. As long as code is linked against the `/CONFIDENTIAL` library, and sensitive data sources are identified, the library ensures that sensitive data does not accidentally leak from enclaves (a property called information-release confinement). `/CONFIDENTIAL` achieves this goal by restricting enclave communication with external memory through a narrow interface.

The key difference between `/CONFIDENTIAL` (and also Moat and VC3) and EnGarde is in the threat model—in `/CONFIDENTIAL`, the client compiles his code against the the `/CONFIDENTIAL` library, but the cloud provider does not check that the code has been compiled against this library. Thus, `/CONFIDENTIAL` is developed for the benefit of the client. In contrast, EnGarde focuses on mutual trust and SLA compliance. With EnGarde, the cloud provider can check that the client has compiled his code against a library such as `/CONFIDENTIAL`. The cloud provider therefore obtains an assurance that the client's code is policy-compliant. However, he does not learn any further facts about the client's code, thereby protecting client confidentiality.

Intel SGX does not protect applications against side-channel attacks and EnGarde also does not attempt to eliminate this attack vector. Yuanzhong *et al.* [41] demonstrate that by exploiting the fact that page table management in SGX is under the control of the OS, a malicious OS can manipulate page tables and page faults to learn memory access patterns of an enclave and therefore can infer private information of that enclave. Similarly, AsynShock [39] controls page access permissions of a multi-threaded enclave-based application to exploit synchronization bugs that might lead to memory corruption or crashes. It offers a tool to widen the attack window in synchronization bugs by interrupting a thread by removing the read and execute permissions from enclave pages and then scheduling another thread whose execution causes synchronization bugs.

Finally, recent work on Ryoan [16] has leveraged the Intel SGX to build a sandbox for distributed applications. Like EnGarde, Ryoan also relies on NaCl [42] to enforce

restrictions on code loaded inside an SGX sandbox, but does so for an entirely different purpose. While Ryoan uses NaCl to ensure that code loaded into an enclave only has restricted control transfers, EnGarde uses NaCl only for reliable disassembly.

Recognizing Functions in Binary Code. EnGarde assumes that client binaries are shipped with symbol-table information (binaries that do not contain this information are auto-rejected by EnGarde). This helps identify functions in binary code which might be needed by the policies on verifying the binaries. Recognizing functions in COTS programs which do not contain debug information has become a growing interest in recent years. For instance, both supervised machine learning [27] and neural network-based algorithms [35] have been applied to recognize functions in stripped binary executables. However, these approaches are still in their infancy, and cannot guarantee 100% accuracy [35]. As these techniques develop and improve in their accuracy and performance, EnGarde can be enhanced to even consider stripped binaries as enclave code.

Instrumenting Code to Thwart Attacks. For years, various solutions have been proposed to defend against control flow hijack attacks due to software bugs. These include stack canaries to protect return addresses and other control data on the stack [13] uses stack canaries, and various forms of control-flow integrity protection (*e.g.,* [12, 24, 37, 38]) use binary rewriting to enforce CFI protection. Cloud providers may require clients to compile their code with such instrumentation. As we saw in this paper, EnGarde can accommodate a variety of policy modules that check that enclave code has been instrumented as agreed-upon by the cloud provider and the client.

7. Conclusion

This paper presents the design and implementation of EnGarde, an enclave inspection library that preserves the security benefits offered by the SGX and allows the cloud provider to verify the client's SGX-based enclave against a set of policies mutually agreed by the cloud provider and the client. In EnGarde, the cloud provider and the client mutually trust the inspection library with policy enforcement. EnGarde achieves its goal by using SGX's hardware attestation and having an encrypted channel set up between the cloud provider and the client. EnGarde only allows the client content to execute if the content follows mutually agreed policies. We have evaluated the effectiveness of EnGarde by using it to enforce three popular security policies for various real world applications.

Acknowledgments. Funded in part by NSF CNS-1420815 and a gift from Microsoft Research India.

References

[1] The GNU C library (glibc). https://www.gnu.org/software/libc/index.html.
[2] Intel software guard extensions (Intel SGX) SDK. https://software.intel.com/en-us/sgx-sdk/download.
[3] musl libc: standard C/POSIX library and extensions. https://www.musl-libc.org.
[4] QEMU: a generic, open-source machine emulator & virtualizer. http://qemu.org.
[5] Intel64 and IA-32 architectures software developer's manual volume 2 (2a, 2b, 2c & 2d): Instruction set reference, A-Z, 2016.
[6] SGX protected memory limit in SGX, 2016. Platform and Technology Discussion, Intel Software Guard Extensions (Intel SGX), https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/670322.
[7] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP Workshop*, 2013.
[8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM TOCS*, 33(3), 2015.
[9] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: Confidential zookeeper using Intel SGX. In *ACM Middleware*, 2016.
[10] S. Davenport and R. Ford. SGX: The good, the bad and the downright ugly. In *Virus Bulletin*, January 2014. https://www.virusbtn.com/virusbulletin/archive/2014/01/vb201401-SGX.
[11] U. Drepper. How to write shared libraries. https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf.
[12] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *ACM/USENIX OSDI*, 2006.
[13] U. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, 2010.
[14] R. Sinha *et al.*. Moat: Verifying confidentiality of enclave programs. In *ACM CCS*, 2015.
[15] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP Workshop*, 2013.
[16] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *ACM/USENIX OSDI*, 2016.
[17] Intel. Intel Software Guard Extensions programming reference (rev. 1) #329298-001US, September 2013.
[18] Intel. Intel Software Guard Extensions programming reference (rev. 2) #329298-002US, October 2014.
[19] P. Jain, S. Desai, S. Kim, M. Shih, J. Kee, C. Choi, Y. Shin, T. Kim, B. Kang, and D. Han. OpenSGX: An open platform for SGX research. In *NDSS*, 2016.
[20] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Tech. Conf.*, 2014.
[21] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *HASP Workshop*, 2016.
[22] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Shevegaonkar. Innovative instructions and software model for isolated execution. In *HASP Workshop*, 2013.
[23] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *PLDI*, 2009.
[24] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM CCS*, 2013.
[25] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, 2016.
[26] T. Roeder. LLVM - Add forward-edge control-flow integrity support. https://reviews.llvm.org/D4167.
[27] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI Conference*, 2008.
[28] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 1), August 2013. http://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intelsupcoming-software.html.
[29] J. Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 2), September 2013. http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intelsupcoming-software.html.
[30] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.
[31] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P*, 2015.
[32] SCO. System v application binary interface, intel386 architecture processor supplement. http://www.sco.com/developers/devspecs/abi386-4.pdf.
[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Tech. Conf.*, 2012.
[34] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *1st ACM Intl. Wkshp. on Security in SDN and NFV*, 2016.
[35] E. Chul Richard Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, 2015.
[36] R. Sinha, M. Costa, A. Lal, N. Lopes, S. Rajamani, S. Seshia, and K. Vaswani. Design & verification methodology for secure isolated regions. In *PLDI*, 2016.
[37] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *USENIX Security Symposium*, 2014.
[38] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P*, 2010.
[39] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronization bugs in Intel SGX enclaves. In *ESORICS*, 2016.
[40] B. Xing, M. Shanahan, and R. Leslie-Hurd. Intel software guard extensions (Intel SGX) software support for dynamic memory allocation inside an enclave. In *HASP Workshop*, 2016.
[41] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.
[42] B. Yee, D. Sehr, G. Dardyk, J. Bradley Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.
[43] Y. Yu. Intel software guard extensions for Linux OS, 2016. https://01.org/intel-softwareguard-eXtensions.