1. Implementation

We implemented a proof-of-concept of the system on top of OpenSGX [?], an open source Intel SGX emulator. The system supports x86-64 ELF executables which are compiled as position independent executables (PIEs) and are statically linked. In this section, we first describe the modification we made to OpenSGX. We next discuss the implementation of the bootstrap, the decoder and the loader.

1.1 OpenSGX

OpenSGX [?] is implemented atop of QEMU's user mode emulation. It also provides an emulated operating system layer that implements services such as executing privileged SGX instructions, an enclave program loader, a user library that helps minimizing the attack surface between the enclave and the host program via the use of trampoline and stub. The framework also offers debugging support and performance monitoring.

1.2 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. Therefore we change the default number of EPC pages from 2000 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 300 to 5000.

1.3 Code Provisioning

The loader first generates a 2048-bit RSA key pair and then establishes a socket connection to the client machine. As a next step, the loader 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. The loader then receives encrypted chunks of the executable the client wants to run inside an enclave. It then decrypts and assembles the chunks to form a complete in-memory representation of the executable.

1.4 Binary Disassembly

One common challenge in disassembling a binary is mixing of code and data within the code section. We assume 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 the Google Native Client [?], 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 which are represented byte a structure including the sequence of bytes defining the instruction and various metadata information such as the number of prefix bytes, number of opcode bytes, number of displacement bytes [?].

The disassembler of the Google Native Client does not keep track of all disassembled instructions. Instead, during the disassembling process it uses a buffer that stores the most recently disassembled instructions [1]. This stems from the fact that the Google Native Client 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 structure.

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.

1.5 Loading

After the executable has been checked and confirmed to follow certain policies the loader next loads the executable. In particular, it maps the text segments, data segments and bss segments to the enclave memory, making the text segment be executable but readonly, 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 finally sets up a call stack and transfer control to the executable.

2. Evaluation

In evaluating the system, our main goals were

- To demonstrate various policy checks that are enabled by the system; and
- To understand the performance overhead introduced by the components of the system.

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: nginx, memcached, netperf, otp-gen, graph-500 and two benchmarks 401.bzip2 and 429.mcf. In all experiments, all the applications are compiled as position independent executables (PIEs) and are statically linked. Also, to keep the size of the executables small all applications are linked against musl-libc [?] instead of GNU libc [?]. Table 1 shows the lines of code of all the components of the loader.

[1] The current version of the Google Native Client's disassembler stores four instructions

Components	Lines of code
Code Provisioning	270
Loading and Relocating	188
Checking Executables linked against musl-libc	1949
Checking Executables Compiled with Stack Protection	109
Checking Executables Containing Indirect Function-Call Checks	129
Client's side program	349
Musl-libc	90728
Lib crypto (openssl)	287985
Lib ssl (openssl)	63566
Total	453349

Table 1. The lines of code for each component in the loader

Benchmarks	Number of Instructions	CPU cycles of Disassembling	CPU cycles of Policy Checking	CPU cycles of Loading and Relocating
Nginx	262228	694405019	1307411662	128696
401.bzip2	24112	34071240	148922245	4239
Graph-500	100411	140307017	246669796	4582
429.mcf	12903	18242127	123895553	4363
Memcached	71437	137372517	489914732	8115
Netperf	51403	90616563	367356878	18090
Otp-gen	28125	42823024	198587525	5388

Table 2. Overhead of the system when checking different benchmarks linked against musl-libc

Benchmarks	Number of Instructions	CPU Cycles of Disassembling	CPU Cycles of Policy Checking	CPU Cycles of Loading and Relocating
Nginx	271106	719360640	713772098	128662
401.bzip2	24226	34292136	1751276	4206
Graph-500	100488	140588361	195218892	4548
429.mcf	12985	18288921	31459881	4330
Memcached	71677	137877497	325442403	8081
Netperf	51868	91577335	183274713	18057
Otp-gen	28217	43053386	217302816	5355

Table 3. Overhead of the system when checking different benchmarks compiled with stack protection

Benchmarks	Number of Instructions	CPU Cycles of Disassembling	CPU Cycles of Policy Checking	CPU Cycles of Loading and Relocating
Nginx	267669	821734999	20843253	128668
401.bzip2	24201	34235817	1751276	4206
Graph-500	100424	140429738	7014913	4548
429.mcf	12903	18242127	1177429	4330
Memcached	71508	138231446	5301168	8081
Netperf	51431	91161601	3775318	18057
Otp-gen	28132	42829680	2334847	5355

Table 4. Overhead of the system when checking different benchmarks compiled with indirect function-call checks

2.1 Linked against a Required Library

When a cloud provider allows a client to run code on her platform, she often expects the client to run some particular versions of the code, for example a version of the code with security updates. As a result, there is a need for the cloud provider to checks if the client has provided a required version of the code.

We implemented a policy check that verifies whether an executable is linked against musl-libc [**?**] version 1.0.5. To perform this check, we first generate the SHA-256 hashes of all the functions of the required version of musl-libc. The policy check starts by iterating through the instruction buffer and looking 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 computes the SHA-256 hash of all the instructions starting from the beginning to the end of the function. To determine whether an instruction is the end of the function, we rely on the fact that all the instructions after the beginning of the function to the end of the function are not at the beginning of a function and therefore do not exist in the symbol hash table. 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.

Similar to [**?**], we 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 [**?**] 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. Table 2 presents the results of our experiments when running this policy check against different benchmarks.

2.2 Compiled with Stack Protection

clang gives the option to emit extra code for checking potential buffer overflows similar to that of GCC [**?**]. Clang's *-fstack-protector* flag lets the compiler to add a guard variable when a function starts and checks the variable when a function exits. For example, when compiling with the flag, the following extra code will be 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 *-fstack-protector-all* which is similar to -*fstack-protector* except that all functions are protected. To check whether an executable is compiled with this flag, the policy check 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 (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 checks that right before 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.

Table 3 presents the results of our experiments when running this policy check against different benchmarks.

2.3 Compiled with Indirect Function-Call Checks

Protecting applications against control flow 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 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 (IFCC) [**?**]. IFCC protects indirect calls by generating 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, 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 apply an IFCC patch [**?**] to LLVM and clang. To check whether an executable is compiled with FFCI checks, the policy first figures out the range of the jump table by relying on the fact that all FFCI jump table entries have the following format:

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

The policy check continues by iterating through the instruction buffer and looking for indirect function calls. It then verifies that before the indirect function calls, there is a sequence of *lea* (lea 0x85c70(%rip),%rax in the above example), *sub* (sub %eax,%ecx), *and* (and $0x1ff8,%rcx) and *add* (add %rax,%rcx). It then computes the target of the indirect call and verifies that the target is within the range of the jump table.

Table 4 presents the results of our experiments when running this policy check against different benchmarks.