Avoiding Leakage and Synchronization Attacks through Enclave-Side Preemption Control *

Marcus Völp, Adam Lackorzynski‡, Jérémie Decouchant, Vincent Rahli,
Francisco Rocha, and Paulo Esteves-Verissimo
CritiX Group — Interdisciplinary Center for Security, Reliability and Trust (SnT)
University of Luxembourg, L-2721 Luxembourg
‡ Kernkonzept GmbH and Technische Universität Dresden
01062 Dresden, Germany
<name>.<surname>@uni.lu, adam.lackorzynski@kernkonzept.com

ABSTRACT

Intel SGX is the latest processor architecture promising secure code execution despite large, complex and hence potentially vulnerable legacy operating systems (OSs). However, two recent works identified vulnerabilities that allow an untrusted management OS to extract secret information from Intel SGX's enclaves, and to violate their integrity by exploiting concurrency bugs. In this work, we re-investigate *delayed preemption* (DP) in the context of Intel SGX. DP is a mechanism originally proposed for L4-family microkernels as disable-interrupt replacement. Recapitulating earlier results on language-based information-flow security, we illustrate the construction of leakage-free code for enclaves. However, as long as adversaries have fine-grained control over preemption timing, these solutions are impractical from a performance/complexity perspective. To overcome this, we resort to delayed preemption, and sketch a software implementation for hypervisors providing enclaves as well as a hardware extension for systems like SGX. Finally, we illustrate how static analyses for SGX may be extended to check confidentiality of preemption-delaying programs.

CCS Concepts

•Security and privacy → Hardware security implementation;

Keywords

information-flow, SGX-enclaves, microkernels, preemption

1. INTRODUCTION

Over the last decades, several system architectures advocated the co-hosting of rich functionality (e.g., provided by a legacy operating system (OS)) next to security critical applications. For example, microhypervisor-based systems provide a limited but more trustworthy execution environment next to virtualized or para-virtualized legacy OSs [7].

Intel SGX [2] is the latest generation of these solutions. Rather than relying on a hypervisor, hardware modifications and a hypervisor-like implementation in microcode allow a non necessarily trustworthy management OS to construct execution environments—*enclaves*—that benefit from extended security guarantees. To launch an enclave, the management OS marks a subset of the system memory as belonging to the enclave. SGX hardware then protects the confidentiality of stored data, and checks integrity and freshness. The management OS may retrieve this memory, but while the enclave exists, it can only return a page with exactly the same content as retrieved. The mechanisms used to ensure these security guarantees consist in tagging and encrypting all cachelines of an enclave prior to writing them back from the processor caches to main memory.

Unfortunately, the current version of SGX fails to implement this protection in a tamperproof manner as has been demonstrated in two recent publications: (i) Xu et al. [14] managed to extract significant amounts of confidential data (whole images) from enclaves by carefully paging enclaves to observe their control flow; (ii) Weichbrodt et al. [13] carefully controlled the scheduling of enclave threads to trigger concurrency bugs in enclave-protected applications. Both attacks come from the fact that SGX allows the management OS to preempt the execution of enclaves at arbitrary points in time. Although the enclave state is encrypted, this allows the management OS to observe the addresses of accessed code and data with cacheline granularity, and reveals some further changes in the enclave state.

To avoid this problem, an obvious solution is to place enclave resources under the control of a trustworthy OS. However, this OS would then have to be tamperproof to enforce the security of enclaves. In contrast to the functionality required to create enclaves[1], the algorithms governing enclave

*This work is in part supported by SnT — University of Luxembourg and Fonds National de la Recherche Luxembourg through PEARL grant FNR/P14/8149128.

[1] This work is primarily concerned with enclave resources being under control of an untrusted management OS. We therefore do not limit our approach to either hardware enclave providers (like SGX) or software solutions like a microkernel plus Inktag [6], but discuss both variants.

To appear in ACM SysTEX '16, Dec. 12-16, 2016, Trento, Italy
Autor version. DOI: http://dx.doi.org/10.1145/3007788.3007794

resource management can become quite complex. Inside the OS, these algorithms would be part of the trusted computing base and a worthwhile target for attacks. In this paper, we investigate how *delayed preemption* (DP) can help foil these attacks while maintaining resource control through an untrusted management OS.

One preliminary remark is that Intel SGX does not guarantee the availability of the enclave, or that it will make progress, unless the management OS is trusted to offer these guarantees. The extension we propose does not change this situation.

DP [11] was first proposed in the context of the L4-family microkernels [8] as a mechanism to replace the disabling of interrupts in application-level locks and paravirtualized legacy operating systems. Rather than preempting applications immediately, the DP mechanism allows applications to signal to the microkernel that preemptions should be deferred until a point in time when the application has moved forward from a critical section and is safe to be preempted. At the same time, the kernel avoids unlimited deferral of interrupt processing by bounding this deferral through a timeout and by dropping applications that exceed this bound. The interrupt triggering this timeout cannot be delayed by the application.

We first focus on protecting data confidentiality, illustrating the possible though impractical leakage countermeasures that protect data confidentiality even if attackers have fine-grained control over preemptions. DP simplifies these countermeasures. Sec. 4 reiterates previous work detailing how DP can be integrated into security type systems (e.g., extending Moat [3]). In Sec. 5, we then discuss how DP hinders the exploitation of concurrency bugs. Sec. 6 discusses DP implementation concerns.

2. IMPRACTICALITY OF LEAKAGE COUNTERMEASURES IN ARBITRARILY PREEMPTIBLE SYSTEMS

Temporarily storing secret information in observable system state, can prove dangerous, if adversaries have fine-grained control over preemption timing of enclave application code. As illustrated by Xu et al. [14], adversaries would then be able to widen the window of vulnerability during which a sensitive application state may be observed. Recapitulating known results on language-based information-flow security, we show that whilst it is in principle possible to construct leakage-free programs in systems with SGX, it is practically infeasible for performance and complexity reasons. We argue that if adversaries may observe the addresses of code and data memory accesses over extended periods in time by controlling preemptions, then no such access must be secrecy-dependent. Indeed, variations in access patterns would be reflected in variations of possibly observable state, which contradicts non-interference [5]. Any secrecy-dependent computation must therefore be casted into publicly observable pieces of code and executed in memory that is not externally observable (i.e., registers or scratchpad memory).

Let us illustrate this point with the help of AES and Osvik's countermeasure against cache side-channel attacks [10], which we recall in the following.

Table-based implementations of AES (e.g., the Linux implementation, which is based on code from Brian Gladman) use eight tables $T_0, \ldots, T_3, T_0^{(fin)}, \ldots, T_3^{(fin)}$, each storing in memory 256 4-byte words to record combinations of the algebraic functions `ShiftRows`, `MixColumns`[2], and `SubBytes`, the Rijndael S-Box. Given input x, AES proceeds at round $(r+1)$ by computing the state $x^{(r+1)}$ by xor-ing the results of looking up these tables at locations determined by the state of the previous round $x^{(r)}$ and by xor-ing these values with the round keys $K^{(r+1)}$, which are expanded from the key k. The i^{th} byte of the initial state $x^{(0)}$ is $x_i^{(0)} = p_i \oplus k_i$, where p_i is the i^{th} byte the 16-byte plaintext block currently processed and k_i the i^{th} byte of the secret key. The state of the first $n-1$ rounds is computed as:

$$
\begin{aligned}
&(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) \\
&\quad \leftarrow T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_0^{(r+1)} \\
&(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) \\
&\quad \leftarrow T_0[x_4^{(r)}] \oplus T_1[x_9^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_1^{(r+1)} \\
&(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) \\
&\quad \leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)} \\
&(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) \\
&\quad \leftarrow T_0[x_{12}^{(r)}] \oplus T_1[x_1^{(r)}] \oplus T_2[x_6^{(r)}] \oplus T_3[x_{11}^{(r)}] \oplus K_3^{(r+1)}
\end{aligned}
$$

Finally, the algorithm produces the ciphertext in the last round by replacing T_i with $T_i^{(fin)}$.

By observing the access timing of cachelines that map to the same set as the tables, adversaries may learn which portions of the table have been accessed and hence what the secret key must be. To prevent this attack, Osvik proposes to mask key-dependent accesses by touching the entire table in cacheline-wide strides so that the adversary cannot distinguish between key-dependent and countermeasure accesses. However, with tools such as AsyncShock, adversaries may find and preempt the application after a key-dependent access has been performed but before the cleanup could proceed and probe the cache to extract key information. Similarly, if the table is loaded first, adversaries may preempt the application after this cleanup, evict all cachelines and then observe a key-based access.

The AES example illustrates a situation where confidential information is temporarily stored in parts of externally observable system state, whether or not cachelines are allocated. It also shows an example of a countermeasure to repair this temporary exposure to information leakage and how fine-grained preemption control can be used to circumvent this countermeasure. We therefore have to conclude that no secret information must be stored in the data accesses to the memory subsystem if the addresses of these accesses depend on secret information.

One may attempt to convert the secrecy-dependent data access into a secrecy-dependent control flow, e.g., by accessing all entries of the table, discarding those that are not required for the round (i.e., all indexes that do not match any of the state bytes $x_i^{(r)}$ of the current round r). However, this may reveal confidential information in the program timing and in the control flow the program takes. In 2000, Johan Agat [1] showed that cross-copying and similar mitigation techniques can transform out timing leaks. Cross-copying consists in inserting the code of branches of secrecy dependent code paths into the respective other path, replacing the original instructions with dummies to ensure both branches

[2]Omitted in $T_i^{(fin)}$

```
 0  mov t_val[0][0] -> R1
 1  mov t_val[0][1] -> R2
 2  mov t_val[0][2] -> R3
 3  mov t_val[0][3] -> R4
 4  for (idx = 0, ..., 256)
 5    mov T0[idx] -> R0
 6    cmp idx, x0(r)
 7    cmov R0, R1
 8    cmp idx, x4(r)
 9    cmov R0, R2
 10   cmp idx, x8(r)
 11   cmov R0, R3
 12   cmp idx, x12(r)
 13   cmov R0, R4
 14 mov R1 -> t_val[0][0]
 15 mov R2 -> t_val[0][1]
 16 mov R3 -> t_val[0][2]
 17 mov R4 -> t_val[0][3]
 18 ...
 19 // similar for T1 ... T3
 20 // compute round withextracted table values
```

Figure 1: Merge of Osvik's countermeasure in the AES subBytes function through conditional moves.

have the same duration. Mantel and Starostin [9] empirically study how well cross-copying and other mitigation strategies reduce side-channel capacity in an uncontrolled setting. The performance overhead of these strategies is significant (up to 125% for cross-copying and 75% for conditional assignment). However, with SGX and assuming access to the binary code is possible, an adversary may observe the addresses of instruction fetches and decode the instructions, as was demonstrated by Xu et al. [14]. The application in the enclave must therefore both equalize the timing and do this with instructions that are executed irrespective of secret values to avoid leaking confidential information[3]. In the following, we show one possible, though complex solution, which ensures data confidentiality without delayed preemption. After that, we illustrate the simplicity of Osvik's countermeasure with DP.

The pseudocode in Fig. 1 sketches our solution, which merges Osvik's countermeasure into the AES code without reverting to temporarily storing secrets and without inducing secrecy-dependent control flows. We assume a register limited machine. Hence the extraction of table values into the buffer `t_val` in Lines 14–17. The main part of the countermeasure are the conditional moves in Line 7, 9, 11, and 13. `cmov` is not guaranteed to access memory if the condition is not satisfied. Hence, register-to-register moves must be used to update the registers buffering the extracted table values. The reads of previous values in Lines 0–3 are not required for this example because in each round all four values will be updated. We include these accesses to show the general pattern: load, register update using a secrecy-dependent conditional move; unconditional store. Compared to 20 memory accesses per table (4 + 16 for the cacheline-stride accesses in case of 64 byte wide cachelines), a single round now requires 256 memory loads per table. Although the pseudocode in Fig. 1 shows that it is possible to construct leakage free code, even if adversaries have fine-grained control over preemptions, its costs and complexity are significant, as can be seen when comparing it with the complete AES round in Fig. 2.

[3]SGX2 reveals only the page-granular addresses. However, because adversaries may probe cache timings, the code of both branches of a secrecy dependent conditional must be located in the same cachline.

```
 1  start_delay_preemptions()
 2  if (!prepare()) goto 1
 3  if (is_preemption_pending())
 4    stop_delaying_preemptions()
 5    yield()
 6    goto 1
 7  compute AES round
 8  access tables in cacheline stride
 9  stop_delaying_preemptions()
 10 if (is_preemption_pending())
 11   yield()
```

Figure 2: Osvik's countermeasure in AES with delayed preemption

Delayed preemption shields temporarily insecure state from observation until the cleanup procedure completes. In the following, we discuss the security properties of this code, which crucially depend on the non-sensitive but sensitive code specific prepare phase executing without preemption, both solicited and unsolicited, to create a state from which it is safe to execute the confidentiality sensitive parts (the AES round and Osvik's countermeasure). Like AES, which proceeds in sensitive rounds, we assume applications to process confidential data in similar sequences of small sensitive sections, interleaved with insensitive code. DP protects these small sections and the leakage countermeasures that re-establish safe-to-observe states.

3. DELAYED-PREEMPTION BASED LEAKAGE COUNTERMEASURES

The control flow of an application can be preempted in two ways: (i) *solicited preemptions* are triggered by the application executing an operation (e.g., a memory operation with insufficient privileges or a system call) that causes the processor to transition control to the operating system. The operating system in turn handles the cause of the preemption and returns to the application (e.g., after resolving the page fault) or schedules another application (e.g., in case of a blocking system call); and (ii) *unsolicited preemptions* occur as a result of device interrupts, inter-processor interrupts or timeouts programmed by the operating system for scheduling purposes. DP defers involuntary preemptions of the latter class but detects also solicited preemptions.

The user interface of DP consists of two flags: the user-writable *delay flag* (d) and the read-only *preemption-pending flag* (p). In addition, the management OS can configure a timeout value by setting the *maximum delay*, which limits how long the current application can defer preemptions and which is readable by the application.

To start delaying preemptions, the application in the enclave sets the d-flag. As long as the d-flag is set, all preemptions (solicited and unsolicited) will set the p-flag to indicate that a preemption is pending. More precisely, if an interrupt is signaled while the d-flag is set, the DP mechanism defers the handling of this interrupt and sets the p-flag to indicate a pending preemption. It then starts a timer to time out latest maximum delay after the first interrupt was signaled in the current delay phase and continues executing the application. Similarly, if the application returns control to the OS while the d-flag is set (e.g., through a page-fault exception), the p-flag is set to indicate this solicited preemption.

For performance reasons, the application is then encouraged to return control to the OS as soon as possible. It can do so by checking the preemption pending flag and yield-

ing to the OS, which resets the timer and allows the OS to process the pending preemptions. However, to prevent a malicious or erroneous application in an enclave from monopolizing the system by never yielding control back to the management OS, the timer will fire at the latest maximum delay after the first preemption signal.

The key insight why DP helps preventing information leakage is in the detection of solicited preemptions and in the deferral of all unsolicited preemptions.

3.1 Avoiding Solicited Preemptions in Sensitive Code

Sec. 2 and the attack by Xu et al. [14] have already shown that page-faults and other solicited exits carry sensitive information about the program. Our approach is therefore to preload the translation lookaside buffers (TLBs), which cache virtual to physical access rights, to prevent page-faults from happening during sensitive code. That is, in the preparation phase, we execute insensitive code that is located in the same memory pages as the sensitive code and read respectively write (without changing the content) all pages that the sensitive code will write to. The important point is that the patterns in which these accesses are performed reveal no confidential information. If necessary, more pages are accessed to mask secret dependent page access.

Now obviously, if this preparation code is preempted, the operating system on the same core as the enclave may change privileges in the page tables and flush TLBs to counteract this protective measure. We therefore have to execute the preparation code non-preemptively using the DP mechanism and restart the entire preparation phase in case a preemption (solicited or unsolicited) has happened. Also, we have to ensure that no self eviction occurs because then the preparation phase or later the sensitive code could evict a translation from the TLB that will later on be required. Notice that this countermeasure is multiprocessor safe because although the OS instances on other cores may modify the page tables, a core-local preemption is required to shoot down TLB entries. But DP defers this preemption like all other preemptions (with the exception of the DP limiting timeout).

Also, we have to make sure that the DP timeout does not fire during the execution of sensitive code. For this reason, the preparation phase has to check whether the timeout value for the core is larger than the worst-case execution time of the preparation phase (assuming all checks succeed) plus the worst-case execution time of the sensitive code. Moreover, we have to require that the DP timeout value maximum delay can only be modified by the management OS on the same core as the enclave (i.e., that maximum delay is a local resource like the TLB).

3.2 Architecture Specifics

In addition to the above checks, specific characteristics of the processor architecture may require further checks in the preparation phase and additional properties. Examples of these include:

FPU: in case the sensitive code makes use of FPUs or other accelerators that the management OS switches lazily, the preparation code must access these devices to trigger the switch during the preparation phase.

Power Management: if the processor allows cross-core power adjustments, the WCET must be a safe upper bound in all power modes. If only local power adjustments are possible, revealing the power state and checking for a power-state dependent timeout value suffices.

Shared Caches: in case deeper cache levels are shared between cores, the enclave provider must ensure that no information is leaked to other cores by observing the allocation in shared caches. Software enclave providers can achieve this through cache coloring or way-locking.

Device Virtualization: a fundamental requirement for our approach to work is that the p-flag is not virtualizable. Otherwise, the management OS would be able to conceal solicited and unsolicited preemptions by virtualizing the DP mechanism. In case the sensitive code needs to access devices, the preparation phase must perform a benign access as well to check whether the access is virtualized (by observing that the p-flag is set upon VM exit).

4. ATTESTING INFORMATION-FLOW SECURITY

For users to believe in the secure execution of an application inside an enclave, the enclave provider has to prove that it is correctly setup and that it actually runs the desired code. The enclave must also ensure that it correctly implements the necessary leakage countermeasures. Only then will users entrust secrets to the enclave and use it for remote processing.

The first two concerns are taken care of by the dynamic root of trust infrastructure built into SGX, respectively by similar secure and authenticated boot schemes of hypervisor-based solutions, which extend these schemes to applications inside enclaves [4]. Although Moat [3] is one approach to address the latter issue, it does not address side-channel attacks.

In 2008, Völp [12] introduced a security type system to check shared-memory programs, which in the following we call *SHM*. We sketch the basic ideas behind SHM and our future plans for extending Moat to attest confidentiality despite cache side-channels. For a more formal treatment, the interested reader is referred to [12].

Like Moat, SHM considers the effect of active adversaries by interleaving adversarial actions in between any two atomic operations of the enclave. Adversaries non-deterministically modify all state they share with the enclave, a modification that Moat calls *havocing* this non-enclave memory. As is common for security type systems, SHM abstracts from concrete values, keeping only security relevant information (e.g., the labels H and L for secret and public information). Both Moat and SHM verify that successfully checked programs never reveal secret information to the management OS, although SHM does not support SGX. Unlike Moat, SHM considers locks and shared objects. That is, while holding the corresponding lock, an application may store secret information in a shared object, provided all entities that share this object are trusted to adhere to the locking policy (i.e., they will not access the object without holding the lock) and provided the application removes the secret information prior to releasing the lock. SHM further supports collaboration between enclaves by recording which information enclaves may have learned from previous accesses to shared objects. In particular, it considers untrusted applications in enclaves, which attempt to reveal this information through the checked enclave. Unfortunately, the isolation provided by SGX is not sufficient to tolerate completely untrusted

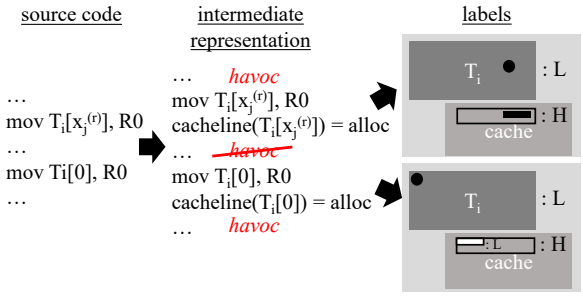

Figure 3: Integration of side-channel effects and assignment of dynamic types

applications. Instead, we have to require for SGX that no application inside an enclave reveals secret information to the management OS. Hypervisor-based implementations are able to tolerate untrusted enclaves.

The type system is applicable to check programs that use the DP mechanism to protect information. Like a lock, delaying preemptions prevents other applications from running on the same core and thus restricts which part of the system state is visible to the adversary. Only if the checked program does not delay preemptions can an adversary see the complete state.

Fig. 3 illustrates our approach on the example of Osvik's AES countermeasure. To consider the effect of side channels, we transform the source code of the enclave program into an intermediate representation where side channels are made explicit. In this example, `cacheline(a) = alloc` denotes that the cacheline corresponding to address `a` is allocated. Since, during the analysis, we do not know the exact value of the AES state $x_i^{(r)}$ and hence the address accessed (the black dot), we transform it into a non-deterministic assignment to any location in the table. Because the address is a secret value (derived from the secret key), the type system changes the labels of the ghost variables, which we use to keep track of the table's cache allocation state, to H. Would the program be preemptible at this point in time (which implies that the cache allocation state would be observable), the type system rejects the program when evaluating the middle havoc operation. It detects part of the cache state to potentially contain secret information. The subsequent table accesses in cacheline stride (shown in Fig. 3 is the first at offset 0) remove any secret information from the table's cache state (shown is the first cacheline as white rectangle). In the intermediate language, this cacheline allocation is an assignment to a constant address, executed in a public branch of the program, which assigns a constant value. Hence, the security type system assigns the label L to the cacheline ghost. DP ensures that the cache state is not accessible before Osvik's countermeasure has removed all secrets. The havoc operation is therefore restricted to those parts of the system state that can be observed respectively modified remotely by other cores. Yielding, respectively the delay aborting timeout makes these variables visible again.

5. CONCURRENCY BUGS

We now turn to the exploitation of concurrency bugs for the purpose of violating enclave integrity as demonstrated by Weichbrodt et al. [13]. The attack proceeded by widening the window of vulnerability during which race conditions

```
thread_1:
1  start_delay_preemptions()
2  free(pointer);
3  pointer = NULL;
4  stop_delaying_preemptions()
5  if (is_preemption_pending())
6    yield()

thread_2:
7  if (pointer) use object;
```

Figure 4: use-after-free concurrency bug

lead to wrong synchronization behavior.

Naturally, DP cannot repair broken synchronization primitives. However, running critical sections under DP protection helps preventing the exploitation of these bugs. We illustrate this on the example of the use-after-free concurrency bug from [13], which we depict in Fig. 4. Thread 1 deallocates a pointer in Line 2 but sets it to `NULL` only in Line 3 to inform the other thread that the object has been deallocated. Consequently, if thread 2 preempts thread 1 in between Line 2 and 3, it would use a deallocated object. With DP, the bug does not vanish as it is still possible for thread 2 executing on a different core to use the object if its check interleaves between Line 2 and 3 of the first thread. However, DP ensures that this window of vulnerability cannot be extended to increase the change of thread 2 hitting this spot. Similar to the example above, DP can help create atomicity wrt. local threads and it may help prevent well known ABA problems (e.g., due to counter overflows) in wait-free synchronization primivites by executing them non-preemptively.

6. IMPLEMENTATION

In this section, we discuss the specifics of implementing DP in a software hypervisor and in SGX-like hardware. Common to both is the communication of the intend to delay using the d-flag, the signalling of solicited and pending unsolicited preemptions with the p-flag and the programming of the timeout to maximum delay once the first preemption is signalled after the d-flag is set.

6.1 Hypervisor-Based Implementation

Let us first consider a software-based implementation of delayed preemption using a trustworthy, enclave-providing hypervisor, which runs the main untrusted management OS.

Before entering preemption-sensitive code, the application in the enclave sets the d-flag to indicate that from now on preemptions should be delayed. One option to implement DP would be to introduce a hypervisor call that disables all interrupts except for the timer interrupt, which it program to fire after maximum delay. However, this involve a potentially costly hypervisor entry and exit every time DP is used. To avoid this overhead, Uhlig et al. [11] proposed locating the d- and p-flag in a per thread user-kernel shared memory location and deferring the detection until the first interrupt (or solicited preemption occurs). In this case, the kernel will catch the preemption signalling interrupt and perform the above operations (i.e., program the timer, disable all other interrupts and set the p-flag), but it will defer the processing of the interrupt until the application returns.

There are several architecture-dependent ways to disable all interrupts with the exception of the timer. If supported by the processor, the hypervisor could mask all interrupts while programming the timer to occur as non-maskable in-

terrupt, as fast IRQ (which on ARM processors can be disabled separately from normal interrupts), or as an otherwise higher prioritized interrupt. For the latter, the hypervisor may raise the interrupt acceptance priority (e.g., by programming the x86 APIC task priority register) to a level larger than all normal device interrupts and IPIs but lower than that of the timer. In case the processor architecture supports none of these options, the hypervisor has to mask all interrupts individually or record all deferred preemptions for later processing.

In addition, the hypervisor must record all solicited preemptions (e.g., the exception, which reflects enclave page faults to the management OS) by setting the p-flag.

6.2 Hardware-Based Implementation

Like a software implementation, a hardware-based implementation has to ensure that neither the application in the enclave has to trust the management OS (to avoid information leakage) nor the management OS to trust the application (to not monopolize the system), but without relying on a hypervisor to context switch DP state.

We propose to integrate the DP mechanism into the per-processor interrupt controller (e.g., the x86 APIC) and to augment the enclave entry and exit code to handle solicited preemptions. As interface, a register suggests itself (e.g., a machine specific register) with op-code aliases for accessing the d- and p-flag at user level. However, there are some intricate details that must be observed to prevent the untrusted management OS from tricking enclave applications into executing preemption sensitive code while being preemptable.

Both the d- and p-flag must be writeable by the application, but not the maximum tolerable delay value, which determines when the timeout fires. Otherwise, if the timeout can be changed by the application in the enclave, it can monopolize the system by setting this timeout to a large value, in particular if this modification becomes effective immediately. Moreover, as illustrated above in Sec. 3.1 and 3.2, maximum delay must be writable only locally by the management OS on the same core as the enclave and the DP register must not be virtualizable.

In addition to the above interface constraints, enclave exit must be adjusted to set the p-flag on enclave exits (i.e., solicited preemptions) if the d-flag is set. Otherwise, the application in the enclave cannot detect solicited preemptions during the preparation phase. This requires that the p-flag is stored and reloaded as part of the enclave-to-management OS context switch. Otherwise, the managment OS could reset this flag upon solicited preemptions during the preparation phase (by itself or, if restricted to enclave mode, through a compromised enclave) and return to the sensitive phase after negating the protective steps.

The yield to the management OS (e.g., in Line 5 of Fig. 2) is not necessary because the preemption signal is directly masked in the interrupt controller before causing a potentially costly kernel entry. Moreover, because the p-flag is located in a device register, the interrupt controller can check pending interrupts when this flag is reset and inject them to the management OS when this flag is cleared. At this time, it may also reset the maximum delay timeout to avoid costly interruption at a later point in time.

7. CONCLUSIONS

In this paper, we have demonstrated how delayed preemption simplifies the implementation of leakage-free code in systems such as SGX that grant adversaries fine-grained control over preemptions. Our hardware implementation of SGX allows applications to implement efficient leakage-preventing countermeasures and the sketched security type system allows proving these countermeasues secure. Our approach is currently limited to core local state only. Directions of future work include further extensions to prevent cross-core side channel attacks and merging Moat with our type system SHM.

8. REFERENCES

[1] J. Agat. Transforming out timing leaks. In M. N. Wegman and T. W. Reps, editors, *POPL 2000*, pages 40–53. ACM, 2000.

[2] V. Costan and S. Devadas. Intel SGX explained. MIT - Techreport, 2016. https://eprint.iacr.org/2016/086.pdf (Accessed: 2016-07-22).

[3] J. S. Denker, S. M. Bellovin, H. Daniel, N. L. Mintz, T. Killian, and M. Plotnick. Moat: a virtual private network appliance and services platform. pages 251–260, 1999.

[4] P. England and J. Loeser. *Para-Virtualized TPM Sharing*, pages 119–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, 1982. IEEE.

[6] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[7] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM.

[8] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 237–250, 1995.

[9] H. Mantel and A. Starostin. Transforming out timing leaks, more or less. In *ESORICS 2015*, pages 447–467, 2015.

[10] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *IACR Cryptology ePrint Archive*, 2005:271, 2005.

[11] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[12] M. Völp. Statically checking confidentiality of shared memory programs with dynamic labels. In *ARES 2008*, pages 268–275. IEEE Computer Society, 2008.

[13] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploting synchronisation bugs in intel SGX enclaves. In *ESORICS*, 2016.

[14] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656, 2015.