Memory Protection Keys: Facts, Key Extension Perspectives, and Discussions

Soyeon Park | Georgia Tech
Sangho Lee | Microsoft Research
Taesoo Kim | Georgia Tech, Samsung Research

Memory Protection Keys (MPK) offers per-thread memory protection with an affordable overhead, prompting many new studies. With protection key extension, MPK provides more fine-grained protection and better functionality. MPK can be an attractive option for memory protection in industry.

Memory safety bugs are commonly introduced by malevolent reads or writes to an unauthorized memory. For example, attackers can read a raw pointer to break address space layout randomization or overwrite a return address on the stack to subvert the control flow of a victim program. These threats lead security researchers to develop various protection schemes to ensure the integrity of control-flow-related data such as return addresses or code pointers on a program's memory. The most promising techniques widely deployed today are control-flow integrity schemes, such as Control Flow Guard (CFG) by Microsoft and Control Flow Integrity (CFI) in the Google Android kernel, but they come with an inevitable performance overhead.

Today, researchers are pondering a memory protection technique beyond control-flow-related data. In April 2014, the OpenSSL cryptography library received reports that the library has a memory leak bug, which can leak server data such as unencrypted user requests, authentication secrets, and private keys from millions of web servers. A lot of commercial websites and their servers are affected by this incident. This incident shows that memory safety bugs that are not related to control-flow-related data also can be critical. In web browsers like Firefox, Write-xor-Execute ($W \oplus X$) protection is enforced to protect just-in-time (JIT) generated code pages, which can be a source of a data-only attack. However, existing memory protection techniques are largely software based and incur unbearable overhead to protect larger memory space because they require expensive operations like changing the permission of memory space.

Recently, hardware-based memory protection techniques have been proposed to fundamentally overcome such performance issues. In 2016, Intel introduced Memory Protection Keys (MPK) for the userspace, called *Intel MPK*. It enables ridiculously *fast* permission change (that is, modifying a register) regardless of the size of the target memory space, which makes it practical to use in real-world applications. Combined

Digital Object Identifier 10.1109/MSEC.2023.3250601
Date of current version: 20 March 2023

 Copublished by the IEEE Computer and Reliability Societies 1540-7993/23©2023IEEE

with the existing page-table-based protection, MPK can even implement *execute-only* memory spaces on a per-thread basis.

One caveat, however, is that MPK provides only 16 protection keys, meaning that one application can have only 16 different groups of memory spaces for protection. To overcome this problem, we proposed libmpk[1] as a general solution to augment the functionality of MPK such that an application can create an arbitrary number of protection domains, for example, protecting the private keys of an arbitrary number of virtual hosts in a web server or protecting a JIT region per each domain in a web browser. Since our initial exploration in this space in 2019, many applications have been proposed to adopt MPK for conventional uses of memory protections, but rather surprisingly, we also see various novel uses of MPK in unexpected areas (for example, detecting race condition).[2,9]

In this article, we look at the current stage of MPK research as well as industry adoptions from the perspective of MPK's *key extension*. Also, we revisit some challenges in adopting MPK in real-world applications and share our views to improve the usability and security of Intel MPK.

MPK Architecture

MPK is a new hardware feature available from Intel Skylake server CPUs. It is similar in concept to the `mprotect` system call in changing the permissions of memory space. However, 1) it allows *fast* permission changes without context switching to kernel space, and 2) it changes the permission of a memory *region* at once. Underneath, MPK keeps track of the read and write permissions of 16 memory regions in a user-accessible register, called a *protection key rights register* (*PKRU*), and provides two additional instructions to read (`RDPKRU`) and write (`WRPKRU`) the permission of each memory region. Since two bits are used to indicate read and write permission, the 32-bit `PKRU` register can represent only 16 different memory regions.

With MPK enabled, access to a memory page is first checked with the permission in a `PKRU` register—the CPU looks up its permission by using a *key*, an index of the `PKRU` register, noted in the page's page table entry. It enables MPK to support execute-only memory. In conventional x86, execute and write permissions imply read permission. Hence, when we set the permission of a page to an executable with `mprotect(addr, len,` and `PROT_EXEC)`, the page is also readable intrinsically. However, with MPK, if both read and write permissions are not set in the `PKRU` register, but the execute permission is set in the page table, then we can effectively implement an execute-only memory (see Figure 1).

It is worth noting that `PKRU` is a register per core, meaning that each thread running on a different core might have different permissions for the same memory region, unlike `mprotect`, which changes the permission of each page globally (that is, for all of the cores). This is the main source of performance overhead—requiring a translation lookaside buffer flush and interprocessor interrupt for all cores—in `mprotect`-based solutions, which is prohibitively nonscalable in a multicore machine. Our past experiment shows that the `WRPKRU` instruction takes 23.3 cycles to update the permission of pages associated with a protection key. However, calling an `mprotect` system call takes 1,094 cycles, which is the traditional way to change the permission of pages.

Moreover, a protection key can be assigned to multiple pages that do not need to be contiguous; however, in doing so, an `mprotect` system call has to update multiple page table entries. The only way to change the permission of sparse pages with an `mprotect` system call is to call it multiple times, which incurs high overhead. In addition, permission updated by an `mprotect` system call is synchronized among threads; thus, it cannot prevent race condition attacks. However, MPK-based memory protection does not synchronize the updated permission with other threads with the result that it does not suffer from race conditions caused by permission synchronization.

Challenges

We visited the hardware primitives and attributes of MPK to protect memory. As we can see, MPK can be a helpful method to protect diverse types of data in memory for real-world applications. However, to apply MPK to real-world applications, a few critical usability and security issues should be overcome.

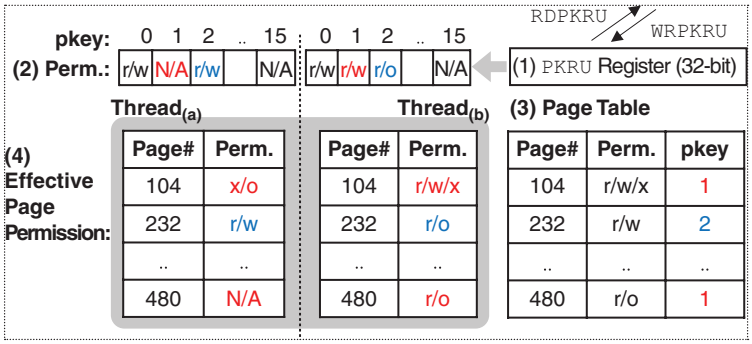

Figure 1. An overview of the MPK architecture. MPK checks the permission of a page per thread according to (1) the PKRU register. The intersection between (2) MPK permissions and (3) page permissions described on the page table determines (4) the effective permissions. Perm.: permission; N/A: not applicable; pkey: protection key; r/w: readable and writable; x/o: executable only; r/w/x: readable, writable and executable.

Limited Hardware Resources

As mentioned previously, MPK supports up to 16 protection keys that are bound to the hardware limitation. Since one protection key can be assigned to multiple and sparse pages, 16 protection keys can manage the permission of more than 16 pages. However, the pages associated with a protection key can only have the same permission for the key, so we call the pages a *domain*.

Sixteen protection keys are enough for some applications as they can build 16 domains, but others may need additional protection keys for more fine-grained protection or just to function. Developers always have a chance to face the lack of protection keys without the expectation as the protection keys are a shared resource for all threads running first-party functions or third-party libraries under the same virtual address space. If these functions and libraries are exclusively and simultaneously using protection keys to prevent others from accessing their code and data, they might need more than 16 protection keys and exhaust the number of protection keys. Unfortunately, enlarging the size of the PKRU register does not resolve this issue magically as it requires additional bits to mark a protection key index in the page table entry.

Threats Against MPK-Based Memory Protection

There are a few known threats breaking the integrity of an MPK-based memory protection system:

1. executing specific instructions or system calls to change the value of the PKRU register
2. bypassing the protection keys when accessing the corresponding pages with system calls that ignore MPK permissions
3. exploiting unintentionally exposed memory interfaces that MPK does not affect.

First, an unauthorized WRPKRU instruction breaks the integrity of MPK-based memory protection as WRPKRU updates the value of the PKRU register in the userspace. In addition, the XRSTOR instruction can modify the PKRU register by setting the corresponding bit in the EAX register as XRSTOR is used to restore the previous execution state (for example, register values) during a context switch. A `sigreturn` system call works similarly to XRSTOR as it restores the previous execution state when returning from the kernel after handling a signal. Therefore, an attacker can alter the value of the PKRU register by locating the manipulated state on the stack and making a `sigreturn` system call. As attackers can modify the value of the PKRU register with malicious intent, MPK-based memory protection should consider prohibiting them.

Second, previous studies[8,10] figured out that a list of system calls is known to ignore MPK permissions when they work and access memory.

- *`process_vm_readv`, `process_vm_writev`, and `ptrace`:* These system calls allow a process to read and write the memory area protected by MPK. `process_vm_readv` and `process_vm_writev` system calls were originally designed to provide interfaces to transfer data between the calling process and the other process. They do respect the traditional page permission of the calling process but ignore MPK permissions so that they bypass the MPK permission check. The `ptrace` system call is used to trace and control another process so that it can inspect and modify the execution state (for example, memory) of the traced process. Reading and writing into the memory under the tracing does not check the page permissions, including MPK permissions.
- *`madvise` and `userfaultfd`:* These system calls allow a process to clear and write arbitrary values to the memory area associated with MPK. `madvise` is designed to notice advice to the kernel, mostly for future possible optimization so that it can free and clear pages. `userfaultfd` handles page faults in the userspace so that attackers can manipulate released but protection-key-assigned pages with a custom page fault handler. As `madvise` frees MPK-associated pages without checking whether or not the pages are associated with MPK, the freed pages trigger a page fault in future accesses and can be overwritten by the custom page fault handler.
- *`brk` and `sbrk`:* These system calls are used to allocate and reallocate memory for the heap. If the MPK-protected memory locates in a heap area, the memory can be deallocated and reallocated by these system calls. In the meantime, the associated protection keys are wiped but the content is preserved, so the memory has a chance to be leaked.

Lastly, Linux provides the `proc` file system, which presents information about processes as a pseudo-file in the file system. In particular, the `mem` file in the `proc` file system establishes a mapping with the virtual memory of a running process; therefore, the attacker can manipulate the virtual memory of the currently running process via standard file I/O operations (for example, `open`, `read`, and `write`). For instance, an attacker reads and writes a crafted payload into the file. Then, it will be reflected in the virtual memory of the running process without checking permissions on the corresponding page, so it bypasses the MPK permission check.

Furthermore, MPK suffers from permission violation from fundamental hardware design, known as *meltdown-pk*. Like other meltdown variants, speculative execution allows for reading data speculatively located in MPK domains. It is mitigated by address space isolation and will be mitigated by hardware support in future processors as well.

How Has MPK Been Used in Research?

Motivated by MPK's fast, domain-wise, and per-thread memory protection, numerous researchers have proposed interesting works, including in-process memory isolation, concurrent garbage collection, and data race detection. This section revisits these works and how they can be improved from key extension perspectives.

MPK's domain-wise memory protection naturally enables in-process memory isolation by separating an application's code and/or data into multiple domains and assigning different keys to them. Table 1 shows the seven recent works using MPK to protect memory possessing multifarious data. ERIM[3] provides a way to isolate sensitive data such as cryptography keys and personal information that memory corruption bugs can leak. MonGuard[5] uses MPK to protect the in-process reference monitor and shared libraries against memory corruption bugs that can leak code and data pages.

NoJitSu[6] applies MPK to isolate code and data pages in the memory used for the JIT compiler. In particular, it separates core JIT data (for example, bytecode and emitted JIT code) into multiple domains and selectively allows a legitimate thread to access or update them based on control flow information. This design prevents malicious threads from corrupting the JIT data, unlike the traditional `mprotect`-based mechanism. To prevent code page leakage, MonGuard and NoJitSu benefit from execute-only memory so that they can execute code pages without read permission.

SGXLock[12] resolves the data access asymmetry in Software Guard Extensions (SGX). A host application cannot access the memory of an SGX enclave, but the SGX enclave can read and write the memory of the host application. SGXLock leverages MPK to prevent an untrusted SGX enclave from malicious leaking and manipulating data in the host application. LIGHTENCLAVE[11] utilizes MPK to create isolated light enclaves inside an SGX enclave. Both SGXLock and LIGHTENCLAVE integrate MPK with SGX to improve its functionality.

FlexOS[14] uses MPK to isolate the compartments of Library OS. The compartments cannot break the integrity of each other and compromise shared data located in the shared memory region. PKRUSafe[15] provides heap isolation for safe and unsafe regions in a memory-safe programming language such as Rust. Rust ensures memory safety at the language level, so it guarantees that a safe code does not have memory corruption. However, it supports unsafe code for efficiency and backward compatibility, which might have memory corruption bugs to corrupt the safe heap. Hence, PKRUSafe disentangles the unsafe heap from the safe heap to prevent unsafe code from accessing the safe heap region.

In addition to the in-process memory isolation, which is the intended usage of MPK, several researchers repurpose MPK's thread-local permission control to solve two different problems: garbage collection and data race detection. These repurposed usages would provide insights for future research.

Table 1. A summary of MPK-based in-process isolation solutions with their protected domains and expected threats.

Name	Protected domain	Possible result of attack
ERIM[3]	Sensitive data (for example, cryptography keys)	Sensitive data leakage
MonGuard[5]	In-process reference monitor and library	Code and data pages leakage
NoJitSu[6]	JIT compiler's memory	Code injection, code reuse, and data-only attacks
SGXLock[12]	Host application outside of SGX enclave	Read and write the memory of the host application
LIGHTENCLAVE[11]	Pieces of SGX enclave memory	Read and write memory of other light enclaves
FlexOS[14]	Compartments of libOS	LibOS's compartments and shared data compromising
PKRUSafe[15]	Heap used in Rust runtime	Read and write heap data owned by the safe region

- *Concurrent garbage collector optimization:* A concurrent garbage collector was proposed to improve the performance of a garbage collector by concurrently running the garbage collector with a mutator so that, ideally, it does not harm the performance of the mutator. Nevertheless, it still degrades the mutator's performance because the garbage collector uses memory barriers to prevent the mutator from accessing objects in the collection area to ensure consistency. Platinum[7] proposes barrier elimination by employing MPK, which is the major cause of incurring overhead in a concurrent garbage collector. Platinum focuses on the fact that different threads can hold distinct permissions on the pages associated with MPK. Platinum splits the heap area into two pieces for the mutator and the garbage collector by assigning them different protection keys. Therefore, the mutator does not require a range check to ensure that the object is not in the collection area. The collection area is protected by MPK and maintains distinguishable permission between the garbage collector and the mutator, so the garbage collector thread can write into the collection area, but the mutator cannot access the area as the writing attempt by the mutator will cause a page fault; it eliminates the necessity of the software barrier.
- *Data race detection:* A data race occurs when two or more threads are running together and trying to access a shared object simultaneously with a read-write or write-write conflict. PUSh[2] and Kard[9] use MPK to detect a data race by locating shared objects in the MPK domain. Using MPK, they ensure that the thread currently holding a lock for the shared object can exclusively access the object. As a result, other threads that are not holding the lock trigger a page fault when they try to access the shared object. Thus, the systems can figure out the existence of a data race if a page fault ensues.

Key Extension Perspectives

Although most of the studies used only a limited number of protection keys, they can provide better isolation if there are more protection keys with zero cost. For example, let us assume a situation in which we want to protect the personal information (for example, home address and social security number) of users using ERIM. In the current situation (using a limited number of protection keys), we have to store all users' personal information in one or a few domains so that an attacker can leak all of the information if there is a security hole in a code region accessing a type of personal information.

However, if we have more protection keys, we can assign one protection key per type of personal information. Thus, even though the code region accessing users' home addresses has some memory leak, it does not leak social security numbers as they are stored and protected in different domains. For similar reasons, NoJitSu, LIGHTENCLAVE, and FlexOS can provide more fine-grained JIT memory, SGX enclave, and libOS's compartment isolation, respectively, with more protection keys.

As PUSh requires the same number of protection keys as the sum of the number of threads and the number of locks by its design, it suffers from the lack of protection keys. To resolve this, PUSh hashes thread IDs and lock addresses and uses the hash value to map to the MPK domain. This can lead to a hash collision, which hides possible data races as multiple objects are assigned in the same MPK domain. This false negative can be resolved if it has more protection keys.

How Has MPK Been Used in Industry?

According to existing research works, it is obvious that memory protection benefits from MPK. MPK has been applied slowly in industry compared to in the research field. In this section, we examine one case in which MPK is applied in industry: Chromium.

MPK Adaption in Chromium

Chromium is an open-sourced web browser used by billions of users around the world. Google Chrome, Microsoft Edge, and many other browsers build on the Chromium project. In 2021, Chromium developers proposed using MPK to protect code pages containing WebAssembly-generated code.

The protection technique is known as $W \oplus X$ *protection*, meaning that the pages conveying executable code cannot have both write and execute permissions at once. Such pages are mostly used for JIT compilers or interpreters as they would emit the compiled code into the executable pages; thus, the pages should have both write and execute permissions. However, if a page has both write and execute permissions simultaneously, the page can be an attack vector as attackers can inject crafted payloads that they would like to execute for malicious purposes.

Chromium originally adopted `mprotect`-based W ⊕ X protection to prevent code injection, but the method incurs a high system call overhead for permission switching as it requires invoking an `mprotect` system call to obtain and revoke write permission. By replacing the `mprotect`-based protection with MPK-based protection, Chromium can alleviate the performance overhead. As an unexpected beneficial effect, it can also protect the memory from race conditions. Figure 2 shows an example in which two threads are maintaining the permission of the WebAssembly memory with MPK to prevent unexpected writing attempts. In the example, the permission of the WebAssembly memory

will be maintained per thread. So, while thread$_{(a)}$ is obtaining write permission with WRPKRU and writing the memory, the writing attempt from thread$_{(b)}$ is denied as thread$_{(b)}$ is unlikely to hold a write permission.

Key Extension Perspectives

The current implementation to protect WebAssembly memory in Chromium spends one protection key. Thus, one protection key can change the permission of the whole WebAssembly memory. This may be safe if a code region that allows a write permission on the WebAssembly pages is small enough to be trusted. However, if the code region contains memory corruption bugs, not only the page where the code region is writing but all the WebAssembly pages are also threatened by the bugs as all the WebAssembly pages are under the same domain bound by the same protection key.

Discussion

Unlike many trials to employ MPK in various applications in research, industry has not yet actively applied MPK. In this section, we discuss the possible reasons that hinder using MPK in industry and in general.

Limited Number of Protection Keys

As mentioned earlier, MPK provides up to 16 protection keys that can create 16 protection domains. This is not enough for some applications, so there are a few works that try to extend the number of protection keys.

libmpk[1] is the first work to extend the number of protection keys in a software-based protection key extension. It works similarly to cache so that it assigns a hardware protection key to the recently used domain. If all hardware protection keys are assigned and no more keys are available, libmpk takes a key away from the least recently used domain. The overhead from the software-based extension is acceptable if the application does not need to change the domain so often but just switches the permission of pages. To switch from a domain for the hardware protection key to another domain, libmpk has to call an `mprotect` system call to update the protection key index located on the page table entry, so it hurts performance.

Hardware-based protection key extension and key extension in aid of an extended page table (EPT) are proposed to ease the overhead from the software-based solution. EPK[13] provides scalable MPK by integrating EPT with MPK. In detail, EPK extends protection keys by using both the EPT index and the protection key index as domain IDs. For instance, EPK allows two domains to have the same protection key but not to be located in the same EPT concurrently. EPK ensures memory isolation between domains in the same EPT by prohibiting duplicated protection keys in an EPT. Thus, all 512 EPTs can create 7,680 (512×15) domains in total, which excludes the first protection key that is reserved for the global domain.

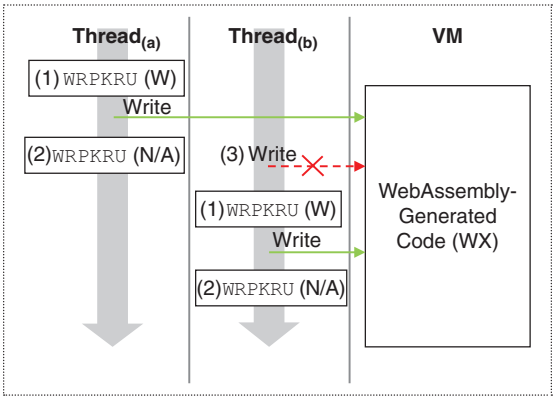

Figure 2. An example showing how MPK works to protect WebAssembly-generated code in Chromium. With MPK, the only code region between two WRPKRUs can access the WebAssembly memory; other accesses will be denied. In the figure, (1) the first WRPKRU sets write permission to the matching protection key, and (2) the second WRPKRU revokes all permissions of the corresponding protection key to prevent (3) further writing attempts. W: writable.

Mitigation for Threats

Although MPK provides a toolset to protect memory, the responsibility to prevent attacks threatening the integrity of MPK-based memory protection is totally upon the developers. A number of aforementioned threats can break the integrity of MPK-based memory protection, so we need to resolve them to use MPK safely.

As WRPKRU can modify the value of the PKRU register, many works using MPK-based memory protection designed a call gate, which is the only legal way to execute the WRPKRU instruction in the binary. WRPKRU instructions outside the call gates are considered illegal; thus, the binary is inspected to detect unauthorized instructions (for example, WRPKRU and XRSTOR). If such instructions are found, the binary is rewritten to eliminate the instructions. Moreover, comprehensive sandboxing to prevent disallowed system calls is necessary as such system calls can modify the value of the PKRU register and bypass MPK protection. Sandboxing filtering specific system calls is widely used, and Jenny[10] proved that their `ptrace` and `seccomp`-based sandboxing incurs affordable overhead (0%–5%) for Nginx.

Supported CPUs

When MPK was just announced to the world, Intel supported MPK only in its server CPUs, excluding

client CPUs; therefore, only server applications could access MPK. However, we see more value in MPK used for client applications. Most of the aforementioned applications using MPK ran on the client side instead of on the server side; for example, Chromium is a web browser that runs on the user's desktop, tablet, and mobile phone.

When we published our software abstraction for MPK in 2019, we received an e-mail from a Chromium developer saying that they want to apply MPK to Chromium. We thought it a good idea in terms of performance benefit and per-thread isolation but stated our concerns that only server CPUs supported MPK back then, so it was not useful to end users. However, Intel started to support MPK in client CPUs at the end of 2020, and AMD CPUs have supported MPK (for example, hardware and instructions) since Zen 3 as well. Thus, MPK is likely to be applied to more client applications in the near future.

Intel MPK provides efficient per-thread permission control on a set of pages. Understanding its hardware primitives and functional characteristics is necessary to fully utilize it for memory protection. Although it suffers from limited hardware resources and possible security threats, we are starting to see studies in research and application in industry using MPK for memory protection, optimizing a concurrent garbage collector, and detecting data races. We also revisited a few studies to overcome the hardware limitations and mitigate the possible threats incurred by specific instructions and system calls. We hope this article encourages more applications to utilize MPK in the real world. ■

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the National Science Foundation Award CNS-1749711, the Office of Naval Research under Grant N00014-23-1-2095, and DARPA V-SPELLS Contract N66001-21-C-4024 and gifts from Facebook, Mozilla, Intel, VMware, and Google.

References

1. S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "Libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 241–254.
2. D. Zhou and Y. Tamir, "PUSh: Data race detection based on hardware-supported prevention of unintended sharing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2019, pp. 886–898, doi: 10.1145/3352460.3358317.
3. A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1221–1238.
4. C. Canella et al., "A systematic evaluation of transient execution attacks and defenses," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 249–266.
5. X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with intel MPK," in *Proc. 13th Eur. Workshop Syst. Secur.*, Apr. 2020, pp. 7–12, doi: 10.1145/3380786.3391398.
6. T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, "NOJITSU: Locking down javascript engines," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.
7. M. Wu et al., "Platinum: A CPU-efficient concurrent garbage collector for tail-reduction of interactive services," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 159–172.
8. R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1409–1426.
9. A. Ahmad, S. Lee, P. Fonseca, and B. Lee, "Kard: Lightweight data race detection with per-thread memory protection," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2021, pp. 647–660, doi: 10.1145/3445814.3446727.
10. D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 935–952.
11. J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A hardware-software co-design for efficient intra-enclave isolation," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 3129–3145.
12. Y. Chen et al., "SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 4129–4146.
13. J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 609–624.
14. H. Lefeuvre et al., "FlexOS: Towards flexible OS isolation," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Feb. 2022, pp. 467–482, doi: 10.1145/3503222.3507759.
15. P. Kirth et al., "PKRU-safe: Automatically locking down the heap between safe and unsafe languages," in *Proc. 17th Eur. Conf. Comput. Syst.*, Mar. 2022, pp. 132–148, doi: 10.1145/3492321.3519582.

Soyeon Park is a Ph.D. candidate in the School of Cybersecurity and Privacy and the School of Computer Science, Georgia Institute of Technology, GA 30308 USA, advised by Prof. Taesoo Kim. Her research interests span software security, with a focus on

automatic vulnerability detection in real-world applications, hardware-assisted memory hardening, and binary analysis with machine learning. Park received a bachelor's degree in computer science and engineering from the Pohang University of Science and Technology. Her research has been supported in part by a Georgia Institute of Technology Institute for Information Security and Privacy cybersecurity fellowship. Her papers have been published in several conferences, including ACSAC, ACM CCS, IEEE S&P, and USENIX ATC. Contact her at spark720@gatech.edu.

Sangho Lee is a senior researcher at Microsoft Research Redmond, Redmond, WA 98052 USA. Prior to joining Microsoft Research, he was a postdoctoral fellow at the Georgia Institute of Technology, and before that, he was a postdoctoral research associate at POSTECH. His research interests include all aspects of computer security, especially in systems, hardware, and web security. Lee received a Ph.D. in computer science and engineering from the Pohang University of Science and Technology (POSTECH), South Korea. He served as a program committee member for several conferences, including ACSAC and USENIX Security. He received the Distinguished Paper Award from USENIX Security 2018. Contact him at sangho.lee@microsoft.com.

Taesoo Kim is a professor in the School of Cybersecurity and Privacy and the School of Computer Science, Georgia Institute of Technology, GA 30308 USA. He also serves as a director of the Georgia Institute of Technology Systems Software and Security Center. He received a Ph.D. in electrical engineering and computer science from the Massachusetts Institute of Technology. Starting from his sabbatical year, he works as a vice president at Samsung Research, Seoul 06765, South Korea, leading to the development of a Rust-based operating system for a secure element. He has published more than 100 papers in top systems and security conferences. He is a recipient of several awards, including the National Science Foundation CAREER (2018) and Internet Defense Prize (2015), and several best paper awards, including USENIX Security'18 and EuroSys'17. Contact him at taesoo@gatech.edu.

Digital Object Identifier 10.1109/MSEC.2023.3271180