Repurposing Software Defenses with Specialized Hardware

Kanad Sinha

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2019

ABSTRACT

Repurposing Software Defenses with Specialized Hardware

Kanad Sinha

Computer security has largely been the domain of software for the last few decades. Although this approach has been moderately successful during this period, its problems have started becoming more apparent recently because of one primary reason — performance. Software solutions typically exact a significant toll in terms of program slowdown, especially when applied to large, complex software. In the past, when chips became exponentially faster, this growing burden could be accommodated almost for free. But as Moore's law winds down, security-related slowdowns become more apparent, increasingly intolerable, and subsequently abandoned. As a result, the community has started looking elsewhere for continued protection, as attacks continue to become progressively more sophisticated.

One way to mitigate this problem is to complement these defenses in hardware. Despite lacking the semantic perspective of high-level software, specialized hardware typically is not only faster, but also more energy-efficient. However, hardware vendors also have to factor in the cost of integrating security solutions from the perspective of effectiveness, longevity, and cost of development, while allaying the customer's concerns of performance. As a result, although numerous hardware solutions have been proposed in the past, the fact that so few of them have actually transitioned into practice implies that they were unable to strike an optimal balance of the above qualities.

This dissertation proposes the thesis that it is possible to add hardware features that complement and improve program security, traditionally provided by software, without requiring extensive modifications to existing hardware microarchitecture. As such, it marries the collective concerns of not only users and software developers, who demand performant but secure products, but also that of hardware vendors, since implementation simplicity directly relates to reduction in time and cost of development and deployment. To support this thesis, this dissertation discusses two hardware security features aimed at securing program code and data separately and details their full system implementations, and a study of a negative result where the design was deemed practically infeasible, given its high implementation complexity.

Firstly, the dissertation discusses code protection by reviving instruction set randomization (ISR), an idea originally proposed for countering code injection and considered impractical in the face of modern attack vectors that employ reuse of existing program code (also known as code reuse attacks). With Polyglot, we introduce ISR with strong AES encryption along with basic code randomization that disallows code decryption at runtime, thus countering most forms of state-of-the-art dynamic code reuse attacks, that read the code at runtime prior to building the code reuse payload. Through various optimizations and corner case workarounds, we show how Polyglot enables code execution with minimal hardware changes while maintaining a small attack surface and incurring nominal overheads even when the code is strongly encrypted in the binary and memory.

Next, the dissertation presents REST, a hardware primitive that allows programs to mark memory regions invalid for regular memory accesses. This is achieved simply by storing a large, predetermined random value at those locations with a special store instruction and then, detecting incoming values at the data cache for matches to the predetermined value. Subsequently, we show how this primitive can be used to protect data from common forms of spatial and temporal memory safety attacks. Notably, because of the simplicity of the primitive, REST requires trivial microarchitectural modifications and hence, is easy to implement, and exhibits negligible performance overheads. Additionally, we demonstrate how it is able to provide practical heap safety even for legacy binaries.

For the above proposals, we also detail their hardware implementations on FPGAs, and and discuss how each fits within a complete multiprocess system. This serves to give the reader an idea of usage and deployment challenges on a broader scale that goes beyond just the technique's effectiveness within the context of a single program.

Lastly, the dissertation discusses an alternative to the virtual address space, that randomizes the sequence of addresses in a manner invisible to even the program, thus achieving transparent randomization of the entire address space at a very fine granularity. The biggest challenge is to achieve this with minimal microarchitectural changes while accommodating linear data structures in the program (e.g., arrays, structs), both of which are fundamentally based on a linear address space. As a result, this modified address space subsumes the benefits of most other spatial randomization schemes, with the additional benefit of ideally making traversal from one data structure to another impossible. Our study of this idea concludes that although valuable, current memory safety techniques are cheaper to implement and secure enough, so that there are no perceivable use cases for this model of address space safety.

Contents

List of Figures

List of Tables

Acknowledgements

This doctorate may not be the hardest thing I have attempted, but it is by far the hardest thing I have achieved. There were times when doubts almost led me to quit, but I am glad I persevered. Many are to blame for getting me past the finish line.

Needless to say, the biggest credit goes to my advisor, Simha. Beyond the wealth of know-how I have acquired from him over the years, his most important lessons have been completely non-technical. He has deeply affected my perception of knowledge, and reformed my outlook towards learning in general. I am deeply grateful for his patience and encouragement throughout, especially in times when research involved just hopelessly searching. If there is one thing I can take away from my time here, I hope it is his ability to have faith in a vision.

I am also very grateful to the other faculty members at the department, particularly Luca Carloni, Martha Kim, Steve Bellovin, and Suman Jana, for the guidance and advice they provided me whenever I sought them. From my numerous discussions and seminars with them, I am humbled by the depth of their intellectual curiosity, and hope to emulate it in my future life. Last but not least, I thank my unofficial “co-advisor”, Vasileios Kemerlis, who, while a graduate student here, found the time and patience to show a novice the ropes in the tricky field of security research.

A lot of friends, especially past and present occupants of CSB 467, have brightened my graduate life at Columbia. In no particular order, my heartfelt thanks to Hiroshi Sasaki, Melanie Kambadur, Paolo Mantovani, Emilio Cota, Andrea Lottarini, John Demme, Adam Waksman, Yipeng Huang, Jared Schmitz, Miguel Arroyo, Joel Porquet, Chester Rebeiro, Kshitij Bharadwaj, Giuseppe Di Guglielmo, Davide Giri, Luca Piccolboni, YoungHoon Jung, Young Jin Yoon, Christian Pilato, Richard Townsend. Vasilis Pappas, Theofilos Petsios, David Williams-King, and others. Of those I did not list, I cannot forget to name Adrian Tang, who motivated and distracted me in the right ways and at the right times, especially when the weight of a PhD proved difficult to bear.

Beyond Columbia, I was also lucky to have a few friends along the way, whose support and help were crucial to get me where I stand now. Of them, I have to mention Alok Sood, who consistently reminded me the meaning of friendship, Courtney Truesdell, who reminded me that much of life is the people in it, and Amit Malik, who was always there even when he was not.

And finally, heartiest thanks to my family, the new ones I found in this country and those I left at home. The warmth of their love (and food) buoys me in times when I am overwhelmed by the life of a researcher. Especially, my parents, my gratitude to whom I cannot even begin to express. In their endless pool of faith in me, I try to see my reflection when my own self-belief wanes. I continue to strive to be the person they have always seen in me.

- Kanad

To Baba and Mum,

and

my good friend, Alok

CHAPTER 1

Introduction

Since the first stack overflow based computer worm was first launched by Robert Morris in 1988, attackers have developed ever sophisticated techniques for bypassing program security, while the defenders continuously try to develop measures to thwart them. Historically, this has resulted in a fascinating trend of cat-and-mouse, so that attackers and defenders are always engaged in an adversarial battle of one-upmanship. However, despite the best efforts of security researchers, attacks are still widely prevalent costing organizations and users significant economic loss. While there are many reasons for this, two major factors have persistently contributed to this state of affairs over the long-term.

First, as sophisticated as attack techniques are becoming, it is trivial compared to the rate at which the complexity of commercial software has grown. For instance, a modern browser includes a Javascript engine/interpretor, PDF viewer, multiple extensions, etc., is written in several languages, and has tens of millions of lines of code. More complexity implies a higher density of bugs, lower test coverage, and a larger attack surface. Pre-deployment detection of errors via testing and validation techniques have come a long way in the recent past with significant research and development being done in the areas of rapid bug detection with tools like AddressSanitizer [100], larger test coverage with various forms of fuzzing, limiting the scope of bugs/attacks with software fault isolation, and so on. Even so, the state-of-the-art in validation lags far behind still, and is nowhere close to accounting for the discovery of all bugs and their scope of exploitation in production software.

Another reason for this is the prevalence of memory-unsafe languages like C and C++, which do not abstract away the address space and allow the program direct access to memory and its contents. Unmitigated access to raw memory means bugs have a higher scope of impact since no checks are built in at the language level to prevent semantically illegitimate accesses. The fact that they have been around for decades also means that there is a significant legacy code base of these languages. Although many type-safe languages have been developed since, prevalence of legacy code and the speed advantage enjoyed by these unmanaged languages has resulted in their continuing popularity among developers in the present and foreseeable future.

In response, security engineers have amassed a rich body of safety measures that aim to counter attacks while being minimally intrusive on the program. Traditionally, these techniques have primarily been implemented at various levels of software ranging from compiler-based static analyses to runtime defensive monitors. Besides some basic hardware support like that for paging, this has remained the status quo for the past few decades, wherein software has borne the brunt of the responsibility for security. However, due to multiple factors, performance being foremost among them, software-based security has not scaled well. As developers stack ever-increasing number of features, while seeking to keep performance overheads at a minimum, it is thus the case that security mechanisms are the ones that get side-tracked or abandoned.

To overcome this problem, hardware designers have lately picked up the gauntlet by providing features that either augment existing security schemes or implement them in its entirety. Employing special hardware for this purpose is highly promising since it potentially mitigates the critical issue of performance faced by software solutions. This has led to proposal of numerous hardware based security solutions in the last decade. Furthermore, as the cost of failure and economic demand for safety rises, hardware vendors have jumped into the fray, announcing numerous instruction set architecture (ISA) extensions. Consequently, major design houses like Intel, ARM, and SPARC have introduced features to combat memory errors [3, 48, 53], pointer corruption [97], control flow integrity [27], isolated execution [75, 84], etc. in recent years.

As heartening as this trend might seem, we have to ask ourselves one crucial question: for all the advantages that specialized hardware provides and the number of hardware solutions proposed so far, why are more of them not seen in deployment today or even included in future microprocessor iterations? The answer partially lies in the process of hardware design and the considerations that go into it. Due to the long time-to-market (of the order of a few years typically), longevity of architectural specifications, and relative immutability of hardware, vendors are hesitant in adding new features if their effectiveness on deployment is uncertain. This is more pronounced when the solution in question is complex, so that integrating it into existing designs becomes intrusive, expensive, and difficult to validate. Hence, the fact that most previously proposed solutions have not translated to commercial implementations implies that hardware security researchers have not hit the right trade off of the aforementioned criteria favorably.

To this end, my research has sought to explore and enable the design of hardware features that are simultaneously effective as security measures, while requiring simple modifications to hardware, thus increasing their chances of deployment. Three case studies are presented in this dissertation that look at different approaches to securing program code and data against external exploitation. Of the three, two are positive studies wherein the first discusses a technique for securing code against runtime disambiguation and injection, whereas the second discusses a solution for practically securing data against malicious memory corruption and manipulation. The third case study presents a negative result involving a technique that seeks to secure the address space as a whole against unwanted memory disclosures, but the implementation overhead was found to be too high, thus making it impractical. With these studies, I hope to motivate the central thesis of this dissertation that asserts the design of low overhead, low complexity, yet effective hardware solutions to augment system security.

1.1 Software Based Defenses

1.1.1 Why Software?

Software based defensive schemes have been and remain the norm in the computer security, and for good reason. Software is cheap to develop, deploy, and update. Thus, the time from the discovery of a bug, to the development of its patch, to pushing the updates to client systems is fairly quick and the process easy. In fact, most software systems are engineered and maintained to accommodate such operations, with advances currently being made in even live patching of systems without any downtime. This is especially critical for security patches, since some of these bugs can be exploited en masse within a short period (especially for web-facing programs), thus increasing their fallout.

Additionally, since software solutions have access to the high-level semantic information of the program, they are easier to reason about and apply. For instance, consider a simple buffer overflow. At the level of the language, the notion of bounds exists and hence, it is easier to determine when they have been violated. This information is absent at the machine level where all that is visible is a flat memory space. Hence, a defense against overflows is potentially easier to implement at the language level, than anywhere below. Consequently, a whole slew of compiler-based software tools have been developed [38, 39, 62, 81] to ensure memory safety for C-based programs. In fact, even hardware based defenses often rely on semantic information passed down to enforce bounds more accurately [36, 79, 105].

A third reason is probably just awareness and the lack of efficient collaboration among software and hardware researchers. Software bugs are easier fixed by people who understand software and how/what attack vectors are/can be utilized to exploit them. Traditionally, software developers have not had a good understanding of hardware microarchitecture or at least, do not have an open channel of communication with hardware developers. Vice versa applies for hardware developers, who are generally unaware of security concerns in software, and do not necessarily have the "hacker mindset". They have only been concerned with power and performance which is what the market has demanded of them thus far. The ramifications of this lack of awareness among hardware designers was made painfully clear with the discovery of the Spectre [60] and Meltdown [68] attacks, which are a direct consequence of some fundamental tenets of out-of-order processing, namely speculative execution. Thus, the insular nature of software and hardware development has contributed to an absence of cross-disciplinary approaches to mitigating persistent and potentially forward-looking threat vectors effectively.

1.1.2 Can it Keep Up with Attacks?

For the past several decades, semiconductor technology has progressed in accordance with Moore's law, giving users exponential increments in performance year-on-year without any modifications to code whatsoever. This blanket speedup of general purpose computation allowed increasingly complicated and feature-rich applications to be deployed with nominal to no overhead apparent to the user. Software-based security also benefited from this phenomenon; as these defenses became more complex and hence, more compute- and/or memory-intensive, their overhead could be accommodated due to faster execution speeds. However, as Moore's law comes to an end, other avenues for speeding up selective aspects of execution are now being explored (multicores, accelerators, etc.). Although some specialized hardware modules for security have been developed (TPMs [117], for instance), this new trend has had little to no positive effect on software security. As a result, users are hesitant to employ software defense schemes that may be secure but slow down the system noticeably. This is undesirable since contemporary attacks have evolved in sophistication to the point that current defenses are not effective enough. Attackers are not only exploiting the previous attack vectors in intelligent ways, but are also continuously inventing novel ones. For instance, if we look at the root causes for remote code execution in Microsoft software in the last few years [125], we see that not only are the classic memory attacks still valid and as relevant, but new vectors like type confusion are exploding. With so many diverse threats to protect against, software based solutions struggle to meet the requirements of performance while still being effective.

Furthermore, there is another reason that limits the effectiveness of some software based solutions — privilege. Most of these solutions operate at the same privilege level, indeed in the same address space, as the program. Any attacker advanced enough to have the capability to affect the program, can, hence, affect the security measure as well. Stack canaries, for instance, have been shown to be easily bypassable via simple brute forcing and/or disclosure of canary value [93].

1.2 Hardware Based Defenses

Due to the above shortcomings of software based defenses, security engineers have started looking towards hardware for assistance, thus creating a new paradigm for defensive techniques. Doing so, however, is not as simple as it seems. In order to develop truly secure and practical solutions in hardware, it behooves us to understand the trade offs and considerations that guide their design.

1.2.1 Why Hardware?

As discussed above, the foremost argument for implementing defenses in hardware is, unsurprisingly, performance. In fact, task-specific special purpose hardware is especially commonplace nowadays with modern commercial system-on-chips containing multiple accelerators for common applications like graphics and cryptography. Specialized hardware for a specific task is generally better than an equivalent software implementation in terms of performance and energy, when the task has a well-defined data set and is self-contained. The same principle applies for hardware based defenses as well – if the solution involves operations being performed on clearly disambiguated data operands, it might be a good candidate for hardware implementation. Consider, Softbound [81], a compiler based bounds checking solution aiming to provide spatial memory safety, which checks the validity of every memory access against the bounds defined in the metadata for the pointer being dereferenced. These metadata are, in turn, stored at a known fixed offset from the pointer. The hardware implementation of the same, Hardbound [36], brings down its overhead from 79% to only 5%.

Additionally, there is also the advantage of privilege. Since hardware is at the root of all trust in the system and has the higher privilege than any piece of software, defenses employed in hardware are harder to disable, manipulate, and/or bypass[1].

One general concern about hardware based solutions, however, is that they are too low-level and hence, do not observe enough semantic information about the program to effectively detect anomalous behavior. This does not always have to be the case, especially when the attack involves a clear violation of architectural principles that reflect common program behavior (e.g., functions should always return to the instruction after the corresponding call to it). Solutions to more complicated problems have taken the following two approaches so far. Firstly, a hardware-software co-design has been shown to be very effective wherein the hardware takes hints from software, either at runtime or through metadata generated through compile-time analyses. Numerous past works, especially in the area of memory safety, have taken this approach since it was observed in many cases that identification of linguistic constructs like pointers and data structures completely in hardware was imprecise and resulted in false positives/negatives [36]. Simple hints/annotations at the program level, when possible, easily and efficiently mitigated this problem. The second approach is a more recent one wherein the hardware infers semantic information via microarchitectural events. This usually requires training against a golden model so that malicious behavior is signaled whenever this expectation is violated. Although the latter technique has only been shown to be effective for anomaly detection [34], it can potentially be used for detecting more specific attack scenarios.

1.2.2 But It is No Silver Bullet

With all the benefits of hardware based defenses outlined above, the question we need to ask ourselves then is, why not implement all or most defenses in hardware? Or more specifically, where do we draw the line at which point it would make sense to implement a software technique to hardware? To answer this question, we have to understand the various factors and trade-offs that

[1]This is a double-edged sword since hardware attacks can also be extremely potent. The recent Spectre, Meltdown, and Rowhammer [59] attacks are cases to this point.

hardware vendors have to contend with for a design to make economic sense.

Performance. As fast as specialized hardware can be, they still usually do introduce some slowdown. As such, engineers have to keep the slowdown figures of the design to a few percentage points for it to be viable (depending on the context). Unless they are able to achieve this, the feature will be deemed fairly slow for most practical purposes. It, thus, becomes a trade-off wherein the vendor has to gauge performance overheads of the feature, how critical that security feature is for the client's application, and what portion of the total user base utilize those applications.

Effectiveness. This is a crucial factor especially for security measures, since, unlike other performance or usability optimizations/extensions, security measures could potentially become quickly outdated if a new attack vector is discovered. Because hardware is not easily updatable, users would just be burdened with a useless, or in some extreme cases, harmful hardware feature, rendering the vendor's investment in the technology unprofitable. Given major hardware design iterations take several years to go from conception to commercialization, vendors, thus, need to be assured of the importance of the problem in question in the years to come, and the effectiveness of the solution against unpredictable vectors within this period.

Complexity. The complexity of any feature directly correlates to the effort and cost of its integration in the final product. This is because complex solution can interact with and occasionally need modifications in multiple microarchitectural subsystems, thus raising not only the cost of development but also that of validation. Since most commercial microprocessors are already extremely intricate, introducing complex features, security or otherwise, entails an arduous effort, and does not guarantee that new bugs will not be introduced. Furthermore, complex techniques often require power hungry structures, thus increasing the energy budget of the chip, especially if it is an always-on feature.

Longevity. Although hardware is not as immutable as most people imagine[2], they are still highly static compared to the flexibility offered by software. Architectural specifications persist over multiple processor generations, users continue to use the same hardware for years on end.

[2]Vendors often release microcode patches to change the functionality of chips on the field.

Tying back to the long-term effectiveness of the measure, it becomes more critical to determine whether the technique exhibits a conservative promise to be a significant deterrent for that attack in the near future. Otherwise, it is hard to justify its inclusion in the final product.

A good hardware solution would, thus, have to assuage all of the above vendor concerns sufficiently and convincingly to make its way into production and sustained usage. Consider Intel MPX [53] for instance. It was released in commercial Intel Haswell processors in 2015, and follows the long line of work in the area of bounds checking support in hardware [36, 45, 79, 80]. This approach required fairly non-trivial changes to hardware and software, and consequently, due to its many issues of compatibility and performance [86], MPX hardly made its way into popular applications. In fact, support for MPX was recently retired in GCC9. Over the last decade, numerous other hardware security features have been proposed in major architectural, systems, and security conferences, and yet only a small handful can be seen in current chips. Even accounting for the fact that, as with other technologies, security ideas need to be improved and cross-validated over multiple iterations before they can be considered refined enough for commercial release, the disparity between research and deployment in hardware security implies a lack of cognizance of the above factors so far.

1.3 Thesis and Contributions

In the light of the above discussion, this dissertation makes the following thesis statement:

Hardware support for software-based defenses can not only complement but also improve their efficacy and performance, without requiring extensive modifications to core microarchitecture.

To support the above thesis, this dissertation makes the following contributions in the form of three case studies.

I. Securing Code. There exist two main attack vectors for compromising the integrity of code and its flow in a program — code injection attacks and code reuse attacks[3] [114]. Both of these

[3]Code pointer corruption might lead to either or both, but we treat it as a data-based attack for this discussion.

techniques have been documented for over 20 years, but are still relevant today. Although code injection was largely mitigated as a primary vector with hardware features like the NX bit and W$\oplus$X, code reuse attacks are still being employed in the wild today. The first contribution of this dissertation is to re-architect a technique called instruction set randomization (ISR) that is considered highly effective against code injection attacks but fundamentally vulnerable to code reuse. In fact with threat models relevant today, even ISR's effectiveness against code injection is questionable. By adding some hardware support, we demonstrate for the first time that our variant of ISR not only prevents code injection attacks under current threat models, but could also effectively counter state-of-the-art code reuse attacks. We achieve this by encrypting code with strong cryptographic primitives statically, and decrypting them just before execution, without adding significant performance overheads. Crucially, as asserted by the thesis of this dissertation, we show how this can be made possible with modifications to the L1 instruction cache and the MMU page walker, thus leaving other core substructures and data caches largely in tact. We also provide a full-system view of this technique highlighting how its protection could be turned on from the very first instruction executed at boot, and the distribution models that need to be in place to accommodate a feature like this.

II. Securing Data. Next in the dissertation, we discuss a novel method for securing data against common types of memory safety violations, which are one of the most long-standing and critical threats in computer security. While previous solutions in this area were effective, they required non-trivial changes to hardware, besides other drawbacks. Our technique, however, introduced hardware support for a simple primitive, i.e., comparison against a large, predetermined random value, and used this primitive to enable practical memory safety in the program with simple software support. Notably, since our primitive itself is so simple, its hardware support is also very trivial (just a comparator at minimum). We also demonstrate how our technique introduces nominal slowdown in the application and is able to provide heap safety even for legacy binaries.

III. Securing the Address Space. The final case study in this dissertation presents a negative result, wherein we modify the virtual address space to present a randomized sequence of addresses to the program instead of the traditional model where addresses follow a linear sequence. This effectively serves to uniformly randomize of the entire address space at a very fine granularity. We argue that doing so prevents the attacker from “traversing” from one data structure to another, thus providing spatial memory safety. It was, however, determined that the implementation and performance cost of this scheme would be too high and would, hence, fail the criteria of our discussion thus far.

CHAPTER 2

Background

Before we dive into the case studies, it is useful to understand the general approaches towards implementation of hardware solutions in order to gain a better insight into the related complexity and sources of overhead, independent of the goal of the individual technique. Towards that end, in this chapter, I will broadly classify past and present hardware security solutions that have been proposed to aid or replace problems traditionally handled by software, and discuss the the nature of modification necessary to accommodate them in a modern general-purpose microprocessor. This classification will be based on the type of hardware implementation, rather than the problem it solves or the effectiveness of different proposals constituting the class. In the subsequent chapters, a detailed analysis of the related work for relevant topics will be presented. Also, note that there are other classes of defenses that counter side channel attacks, provide isolation, etc., and so, are tangentially related to code and data security, but from a hardware or privilege standpoint. Such defenses are considered out of scope for the sake of this discussion.

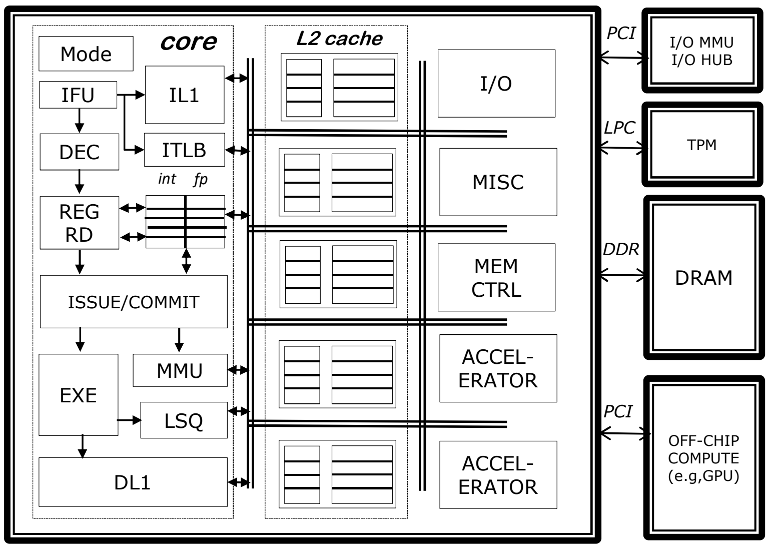


Figure 2.1: Generic baseline architecture and its hardware security mechanisms.

2.1 Baseline Architecture

Prior to discussing various aspects of the cost and benefits associated with particular hardware defenses and the changes they require, we provide the readers with a baseline architectural design against which to evaluate hardware solutions, existing or new.

Figure 2.1 shows the baseline architecture that we use for illustrating all the security techniques. The baseline shows a processor and off-chip components: the off-chip components include DRAM, off-chip compute units which can be GPUs or FPGAs, a TPM and a I/O hub chip which can connects to various peripherals. Inside the processor, without loss of generality, we assume a single core processor but we will point out multicore related issues as they are relevant.

The core includes standard microarchitectural structures: the mode bits determine if the core is in supervisory mode or user mode, the IL1 is the level 1 instruction cache that holds instructions, the ITLB is the translation lookaside buffer which provides virtual to physical address translation for instructions, an instruction fetch unit (IFU), a decoder, a register file that has integer and fp register, an out-of-order issue and commit unit, and several structures for handling memory accesses.

The structures for handling memory accesses include a level 1 data cache, the load store queue

(LSQ) which is used to support reordering of memory requests, MMU which includes the data translation lookaside buffer and different page walking or page walk caching mechanisms. Beyond the core, we assume a L2 cache that is banked, each bank has tags and data. Our baseline chip also includes some special purpose on-chip offload engines (such as a graphics core), a memory controller that reorders and relays request to memory, and I/O controller that relays and manages requests to I/O devices.

All of these on-chip structures are connected by an on-chip network. This baseline is abstract and sufficiently representative to provide intuition on cost/complexity of hardware security mechanisms.

2.2 Semantic Metadata Based Defenses

This class of defenses associates semantic metadata with each pointer in order to ascertain that it is never illegitimately dereferenced. This metadata could take the form of addresses, permissions, and other semantic information regarding the corresponding data structure, and may be stored along with the pointer or in a separate shadow space that reflects the program's memory layout. Works in this category have so far taken one of two forms.

In the first variant, the metadata is used to enforce bounds checking on the pointer at dereference. Each pointer is functionally associated with a corresponding start address and an end address (or length) that specifies the bounds of the pointed structure. In some works, temporal information is also stored that indicates the "version" of the structure, so that if the current structure is deallocated and another allocated in the same region, a pointer dereference of the old structure can be caught. The first hardware support for such a scheme was Hardbound [36], which did not enforce temporal safety. Subsequent works [45, 79, 80] refined this technique to enforce more accurate spatial and temporal bounds. Notably, Intel launched the MPX ISA extension [53] in 2015 that brought this technology into a commercial domain.

The second variant in this class is capability based architectures. Capabilities are a form of

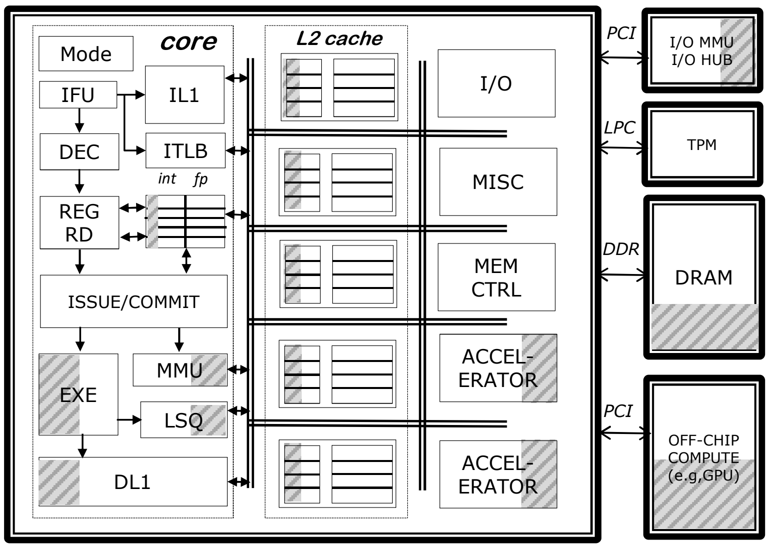


Figure 2.2: Hardware modifications for semantic metadata based defenses (highlighted).

access-control and can be loosely defined as tokens necessary to access a particular resource. Conceptually they can be represented as a data structure consisting of a unique resource-identifier and the associated credential. Privileges are acquired by entities when they receive the appropriate token. These tokens must be unforgeable and are passed to entities, preferably according to the principle of least privileges. Although considered an obsolete technology until recently, CHERI [126] and its follow-up works [22, 23, 123] have revived it, and demonstrated its viability in a practical setup.

Hardware Modifications. Although individual proposals have their own unique design points and optimizations, here we outline common features of a naive design for a typical scheme in this class. Such system conceptually require wider data-paths and memory elements, and access checking circuitry on memory accesses to impose rules dictated by the metadata, which have to be separately accessed. Often techniques provide a separate metadata cache and TLB at the L1, thus functionally extending the existing caches and TLB, to enable fast lookup of metadata. Notably, WatchdogLite handles metadata explicitly just like regular data, while CHERI uses a capability coprocessor a placed next to the pipeline to perform the latter. The hardware units that are affected are shown in Figure 2.2.

2.3 Tagged Defenses

This type of defense is similar to the previous class in that they use metadata (often called tags, in this context), but in this class they are usually just one or a few bits. The tags indicate a state, only meaningful in the context the defense operates. Depending on the solution, these tags could be semantically attached to memory locations, pointers, and/or data. For instance, some techniques [96, 107, 119] attach tags to memory to validate if can be accessed or not. Some [3, 48] go still further by attaching tags to pointers as well, to enforce a degree access control by mandating that tags of pointers and accessed location match.

A more advanced form of this class of defense is dynamic information flow tracking (DIFT) [113]. DIFT uses metadata or *taints* to track the flow of untrusted information within the system, to make sure that it does not unexpectedly affect any trusted or secret portion of the system, or otherwise leak it on untrusted channels. The idea is that if at least one of the operands for an operation is untrusted, the output is untrusted as well and should be limited in its ability to interact with other elements of the program. The same goes for all operations involving this output, and so on. Depending on how the tags are propagated and checked, what gets tainted, and who monitors them, DIFT can be used for a variety of purposes from access control to information leakage.

Hardware Modifications. The degree of tagging support is dependent on what is being tagged and how tags are propagated. Conceptually, the modifications required are similar to the previous class. To detect flow of data through the processor, the data pathways – memory elements (register file, microarchitectural registers, and buffers) and buses/interconnection networks – have to be extended by the tag-width. In practice, however, when the tags are narrow enough, some solutions have leverage/repurpose ECC bits in main memory for tags [3, 48]. Some current solutions aimed at 64-bit architectures also use the unused higher order of the address to store pointer tags, thus significantly lowering the hardware requirements. Depending also on whether the data and tags can be atomically accessed, one or more memory accesses may be necesssary for each data access. Additionally, for DIFT, to taint the output of operations involving tainted data, a small `OR` gate is necessary around computational units which takes as input the taint statuses of the inputs, and marks the output tainted if any of the inputs are tainted. Since both classes are based on per memory metadata, the hardware modifications for this class are similar to that of the previous class shown in Figure 2.2.

2.4 Cryptographic Defenses

Although cryptography has been a popularly used mechanism for data integrity and secrecy, there have only been a handful proposals that utilize it to protect code and data at the process abstraction level. Hence I will give a brief overview of some notable techniques individually below.

ASIST [88] is the first hardware scheme proposed to support ISR (discussed in more detail in chapter 3). Essentially, it encrypts program code with simple XOR or bit transposition. This is done at runtime with a simple key, programmable into a register, which decrypts instructions just before they are stored in the L1 instruction cache. Polyglot [104] operates similarly except it uses stronger AES encryption, besides asymmetrically encrypting the code encryption keys as well. Details of these technique are discussed in more detail in the next chapter.

Recently, a similar randomization technique, called HARD [8] for data was proposed that statically determined equivalence classes of data based on the instruction they were accessed from. These classes were then XOR-encrypted with the same key and the key associated with the instruction(s). Recently, ARM also announced a pointer authentication technology [97], wherein all pointers can be encrypted with a master key and authenticated at dereference. This is to counter out-of-band corruption or forging of original pointers, for instance, through an overflow of a buffer into a neighboring pointer variable.

Hardware Modifications. The techniques outlined above have completely different purposes and hence do not share any common design features except the addition of a decrypting unit. For ASIST and Polyglot, this unit is added at the memory facing interface of the L1 instruction cache and is just a XOR or a large decoder structure. In case of ARM pointer authentication, although design details have not been disclosed publicly, a naive design would introduce an encryption and decryption engine in the pipeline as an additional functional unit. HARD also adds caches around the pipeline and decryption units. Overall, above techniques introduce small overheads on the baseline architecture depending on the specific cryptographic algorithm used and where in the data path the operations occur.

2.5 Logging or Monitoring Based Defenses

Many novel ideas have lately been proposed that leverage microarchitectural logs present in modern microprocessors (hardware performance counters, branch direction stores, etc.) to detect attacks or otherwise anomalous program execution. One of the challenges of utilizing hardware logs is the fact that they may not capture enough information about the execution software, be it the OS or individual programs on it, so much so that any kind of semantic information about the past or present state of execution may not be discernible. To illustrate through a naive example, could one be able to detect that a buffer overflow attack has occurred just by looking at the cache-miss counters? Luckily, modern microprocessors usually maintain a host of different counters or records that have been shown to reflect program behavior fairly well [102]. Depending on the complexity of information sought, some form of heuristic post-processing of these data may be required though, thus requiring an offline software component (running on a separate core, for instance) to process the data and raise the alarm when necessary.

Hardware Modifications. Usually these solutions require very little hardware modifications, since the bulk of the logs are already present in the microarchitecture. Synthesizing this information may however require a small hardware component (in SCRAP [55], for instance), unless this task is offloaded to software [34, 87].

CHAPTER 3

Reviving Instruction Set Randomization with Polyglot

Instruction set randomization (ISR) was originally proposed as a countermeasure against code injection attacks. However, it is largely considered to have lost its relevance due to the pervasiveness of code-reuse techniques in modern attacks, against which ISR is fundamentally ineffective. Additoinally, code injection no longer remains a foundational component in contemporary exploits.

This chapter revisits the relevance of ISR in the current security landscape. We show that prior ISR schemes are ineffective against code injection, but can be made effective against code-reuse attacks, and even counter state-of-the-art variants, such as "just-in-time" ROP (JIT-ROP). Yet, certain key architectural features are necessary for enabling these capabilities. We implement a new ISR system, namely Polyglot, on a SPARC32-based Leon3 FPGA that runs Linux. We show that our system incurs a low performance overhead (4.6% on a subset of SPEC CINT2006) and defends against real-world (JIT-)ROP exploits, while still supporting critical features like page sharing. Polyglot is also the first ISR implementation to be applicable to the entire software stack: from the bootloader to user applications.

3.1 Introduction

Instruction set randomization (ISR) was proposed as a way of mitigating code injection attacks [7, 56]. Specifically, ISR involves *randomizing* the underlying instruction-set architecture (ISA) of a CPU, thereby giving the appearance of a unique instruction set to every program. For instance, the opcode `0xa` may denote `XOR` on one machine, but may be invalid on another. This prevents an unauthorized party (attacker) from using the *same* exploit on multiple targets—any injected (shell)code must adhere to the unique ISA used by the vulnerable program to be effective.

ISR implementations typically "emulate" the random ISA using a layer of encryption. Code is encrypted at the binary level and decrypted on-memory, right before execution. Apart from ASIST [88], the only hardware-based ISR scheme to date, all previous solutions [7, 16, 51, 56, 95] are software-based and primarily utilize dynamic binary instrumentation frameworks, like Pin [71]. This method for generating diversity is far simpler than customizing the decoder on each chip to implement random instruction mappings or changing the microarchitecture of every CPU instance.

However, ISR has a major drawback that impedes its widespread adoption—it is completely ineffective against code-reuse attacks, which is the typical cornerstone of modern exploits [21, 89, 90, 101]. This is because code-reuse attacks, as the term implies, stitch together legitimate code pieces, already present in the address space of a running process. Additionally, as we show later, once an attacker has the means to overcome techniques like W$\oplus$X, ISR can be trivially bypassed as well. Hence, even as a defense against code injection, ISR is no more effective than other established techniques.

Aside from this fundamental limitation, all the previously-proposed ISR schemes also suffer from one, or more, of the following major issues:

① **Unfavorable performance–security trade-offs.** Since instructions are decrypted at runtime, the decryption process falls squarely in the critical path of instruction execution and the associated latency is hard to amortize. Hence, weak encryption schemes are traditionally used to offset this overhead, resulting in new attacks [109, 124].

② **No self-protection.** Since most previous solutions are software-based, they: (a) expose a

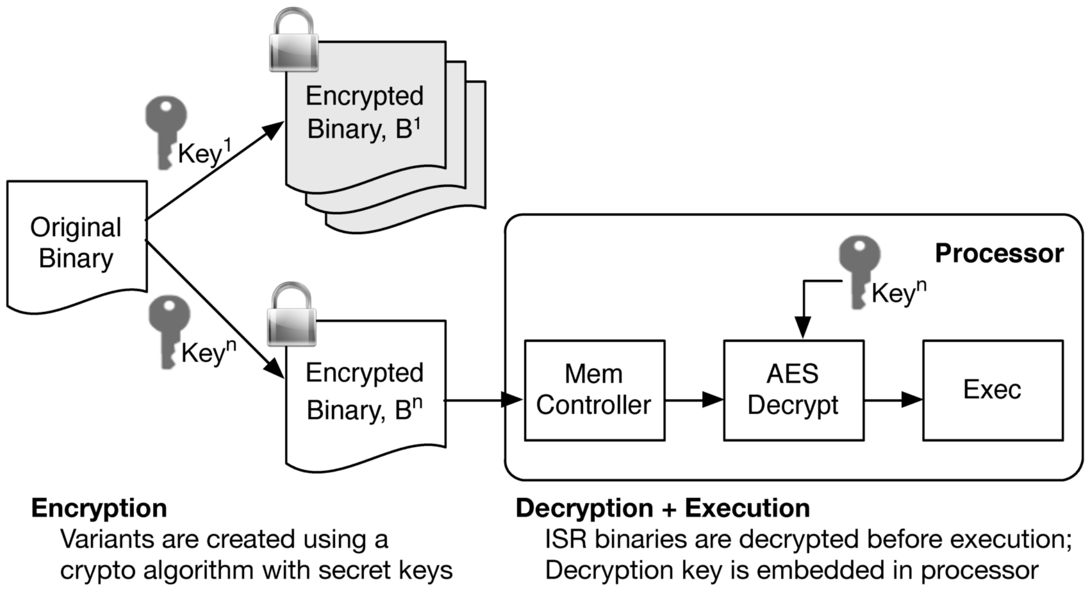


Figure 3.1: High-level overview of Polyglot.

large trusted computing base (TCB), and (b) can be turned off easily, because the enabling framework executes at the same privilege level with the protected application(s).

③ **Incongruent with modern software systems.** Previous schemes provide limited or no support for shared libraries and page sharing. Hence, they are impractical in modern settings, as they incur large memory overheads [6].

④ **Archaic threat model.** Previous approaches do not consider memory disclosure vulnerabilities as part of their threat model. Yet, arbitrary memory reads are a standard component in recent work in the area [5, 29, 33, 46, 106], and quite popular to present-day attacks.

In this work, we propose *Polyglot*, a hardware-based ISR scheme that concurrently addresses all the above concerns. Polyglot not only improves significantly the traditional security properties of ISR, but also counters state-of-the-art code-reuse attacks, which are a novel and more relevant target for this technique. We utilize strong encryption (AES [110] and ECC), successfully overcoming the challenges of impractical performance *by removing decryption from the critical path* with microarchitectural optimizations. We present two schemes: (a) one with no overhead in practical systems, catering to the standard threat model used in anti code-reuse-attacks proposals; and

(b) another with nominal overhead to counter a stronger threat model. Unlike most prior work, we encrypt at the *page*, instead of application, granularity. This not only enables richer diversification, but also allows us to trivially support page sharing and seamlessly apply ISR to the system software (OS, hypervisor, etc.). We are also the first to show how ISR functionality can be logically extended to operate from system boot time, when the memory management unit (MMU), and hence paging, is disabled. Consequently, our implementation of ISR is available from the very first instruction that the system executes.

Figure 3.1 provides an overview of Polyglot. Binaries are encrypted page-wise with AES, while upon execution, the hardware memory controller (with the help of the OS) decrypts the executing instructions as they are transferred into the cache, so that the program remains encrypted in memory. We elaborate on the rationale and motivation of such a design in next section.

In summary, this work makes the following contributions:

- We propose Polyglot, a hardware-assisted ISR implementation, and demonstrate for the first time that it can be used as a defense not only against code injection, but also against (static or dynamic) code-reuse attacks (CRAs) that are a more critical extant threat. In addition, we show how prior proposals fail to fit into this role.

- We overcome the security-performance tradeoffs exhibited by previous ISR proposals, and illustrate how Polyglot can be performant despite using strong cryptographic schemes.

- We uphold the principles of scalable software design by supporting shared libraries and page sharing.

- We implement a SPARC32-based prototype, based on the Leon3 open-source SoC package [66], which allows us to study and demonstrate the full-system viability of our proposal. This enables us to practically evaluate the system from a performance and security standpoint.

3.2 Background and Motivation

In this section, we give a brief overview of previous work on ISR, and claim that in the face of modern exploitation techniques, ISR is completely ineffective. Subsequently, we motivate a novel way in which ISR can become valuable again, when combined with other protection mechanisms, albeit only if strong encryption is employed.

3.2.1 Previous ISR Schemes

Proposal	HW Based	Encryption	Granularity	Shared Libs*	Code Sharing	Self-modifying Code	Scope	Slowdown
Barrantes et al. [7]	✗	XOR	Proc.	✗	✗	✗	App.	High
Kc et al. [56]	✗	XOR	Proc.	✗	✗	✗	App.	High
Hu et al. [51]	✗	AES	Proc.	✗	✗	✗	App.	High
Boyd et al. [16]	✗	XOR	Proc.	✗	✗	✗	App.	Med.
Portokalidis et al. [95]	✗	XOR	Proc.	✓	✓	✓	App.	Med.
Papadogiannakis et al. [88]	✓	XOR, Trans/on	Proc., Kernel	✓	✗	✓	App., Kernel	Negl.
Polyglot	✓	AES, ECC	Mem. page	✓	✓	✓	App., Kernel, Boot-loader	Low

Table 3.1: Comparison of various ISR proposals. (*Shared library support does not necessarily imply sharing them across processes, unless code sharing is allowed.)

Most prior ISR attempts are software based (see Table 3.1), and typically implement randomization using dynamic binary instrumentation tools, such as Pin [71], or platform emulators, like Bochs [64]. Obviously, an attacker can turn off, or subvert, such components as they execute at the same privilege level as the protected application. Apart from other practicality issues, they also exhibit significant slowdowns [95] and were demonstrated to be bypassable [109, 124], due to their use of weak XOR-based encryption.

ASIST [88] sidesteps problems with performance and (the lack of support for) shared libraries by providing hardware support for ISR, and incorporates the best practices of most of the previous techniques. ASIST allows two types of weak encryption: XOR and transposition. The encryption keys are unique to every process, and can be generated either at compile time, in which case the application statically links with all its dependencies, or at load time, where the pages are encrypted dynamically. The latter mode allows shared libraries, but does not allow sharing them between applications, thus incurring a significant (memory) overhead [6]. Absence of page sharing also precludes the copy-on-write primitive, significantly increasing the overhead of `fork`.

3.2.2 ISR against Code-Reuse Attacks

Code-reuse techniques are the attackers' response to the increased adoption of hardening mechanisms, like non-executable memory (NX), in commodity systems. The main idea behind code-reuse is to construct the malicious payload by reusing instructions already present in the address space of a vulnerable process [101]. This powerful technique gives the attacker the same level of flexibility offered by arbitrary code injection, without injecting any new code at all; the malicious payload consists of just a sequence of *gadget* addresses intermixed with any necessary data arguments.

ISR is *fundamentally* ineffective against code-reuse attacks (CRAs), since attackers can construct their payload without knowing how the instructions have been scrambled. One only needs to know the *location* of the appropriate gadgets.

In the presence of various code-randomization schemes [63], state-of-the-art CRAs have been evolved to discover gadgets dynamically, at runtime, by scanning code pages [106]. To prevent this, two conditions have to be met:

① The host binary should differ from the attacker's copy.

② Code should not be readable.

These two conditions are sufficient, as an attacker can neither scan binaries at the host, nor use their own copy in conjunction with the diversified copy to mount a CRA. (Note that this does not rule out data-only attacks [19].)

To this end, we propose combining ISR with a fine-grained (code) randomization scheme to thwart state-of-the-art CRAs. While the latter satisfies condition ① an ideal ISR implementation can also *indirectly* prevent condition ② by revealing (ideally irreversible) encrypted text when code is read.

However, if ISR, plus diversification, is all that is needed, can we just combine code randomization with any of the previous ISR proposals? In other words, can we use ISR based on weak encryption to fulfill the above conditions? Unfortunately, encryption schemes such as XOR and bit transposition can be easily bypassed, even under the presence of fine-grained diversification; during our preliminary experiments we were able to leak keys for the two schemes easily, using entirely architecture- and ABI-independent methods (we elaborate more in Section 3.6.1). Hence, we argue that using stronger encryption, at low cost, is *pivotal* in providing effective ISR-based protection.

3.3 Adversarial Model

We assume that the chip manufacturer is trusted and the attacker is incapable of physically probing the contents of the chip. We also assume that the processor supports some form of W$\oplus$X mechanism (e.g., NX bit). In addition, the OS and the essential system components are otherwise trusted to perform their duties faithfully. Note that side-channel attacks are considered out of scope. We present two versions of Polyglot in this work, which are geared towards two adversarial models.

• **Adversarial Model A.** Under this model, the adversary has access to the source code and/or a non-randomized version of the target program. She is aware of memory corruption bugs in it that can be exploited to gain arbitrary read and write capabilities. The same applies in the kernel setting, but the MMU-related structures are assumed to be protected with existing defenses [32, 122]. Lastly, the attacker also has the ability to carry out a finite number of brute force attacks before being detected. In this scenario, the first variant of Polyglot, namely Polyglot-A, seeks to protect against runtime code-reuse attacks.

This model is identical to previous studies regarding runtime CRAs [29, 46, 106], both offensive and defensive, with the additional extension to the kernel.

• **Adversarial Model B.** Under this model, the adversary does not have access to the source code, but is more capable: she can subvert the OS, including the MMU structures, can access the randomized binary, and has physical access to the system allowing her to snoop/probe the buses, memory, and system peripherals other than the processor internals. In other words, we consider the processor to be the only component in our TCB.

This model clearly goes beyond CRAs and targets (static and dynamic) code leaks in general. In this case, Polyglot-B tries to prevent theft of application keys, and hence, plaintext code.

3.4 System Architecture

In this section, we present Polyglot's architectural design, detailing our software and hardware modifications, and how they inter-operate to achieve our goals.

3.4.1 Software

Binary Generation. To create an "ISRized" binary, we symmetrically encrypt a diversified version of it, at page granularity, with randomly generated keys. (Note that only executable sections are encrypted.) These key-to-address mappings are then asymmetrically encrypted using the target processor's public key and packaged into the binary itself. Since code is encrypted at a page granularity, the executable, and its required shared libraries, possibly encrypted by different sources, are able to interoperate. Additionally, asymmetric encryption ties the binaries to their respective hosts.

Binary Execution. The dynamic loader and the OS are responsible for extracting the encrypted keys from ISRized binaries. In particular, the OS is in charge of them, during its execution lifetime, and for setting up the process' page tables, as well as its own, in a format expected by the hardware. (Note that our scheme allows code pages to exist in plaintext if necessary.) Additionally, since we encrypt at page-granularity, code sharing for shared libraries, as well as for forked processes, is

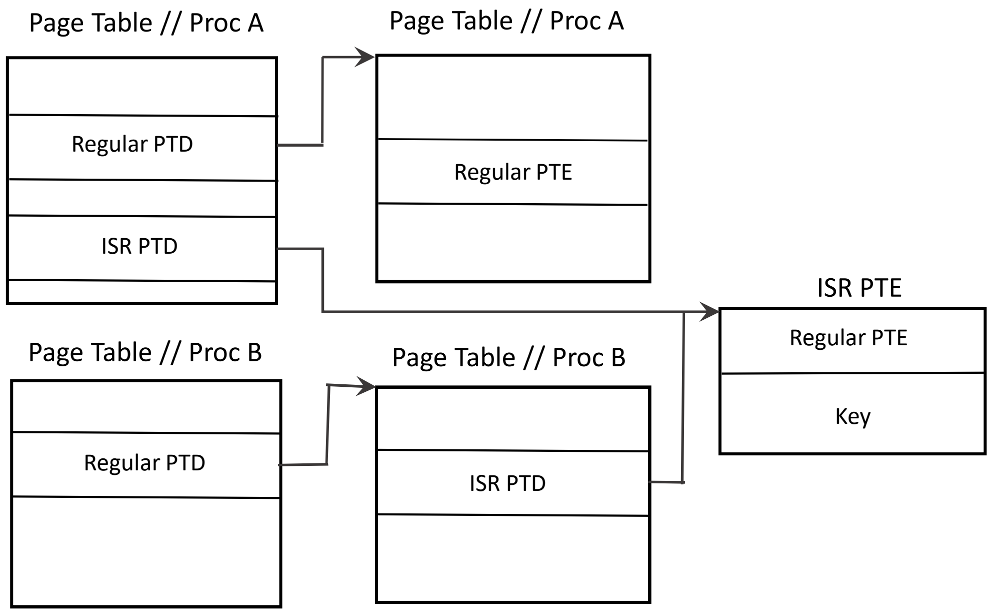


Figure 3.2: Modifications to page table. We use a reserved field in page-table value type field to indicate a subsequent ISR PTE. The ISR PTE here corresponds to a page shared between Processes A and B. PTD indicates a page table descriptor, which is a pointer to the next level, whereas PTE is the final translation.

readily supported, i.e., by using the same translation entry among the page tables of processes that share a particular page (see Figure 3.2).

Supporting ISR at the kernel level is achieved simply by changing the kernel's own page mapping(s) to an ISR-PTE version. Overall, our modifications added $\sim$1100 LOC to the Linux kernel (v3.8.1). Since paging is disabled at system boot, we randomize the bootloader by encrypting the whole image according to its layout in physical memory, so that encrypted execution is enabled from the very first instruction. Care is taken, however, to ensure that when the MMU is turned on, and paging enabled, the respective keys match (i.e., before and after enabling virtual addressing).

3.4.2 Hardware

All previous ISR implementations suffer either from prohibitive slowdowns, weak security, or both. Polyglot however aims to overcome both, and proper hardware support plays a crucial role in this.

Given our threat model, we face quite a few challenges. Firstly, because the attacker can read and modify program memory and can probe system buses, we can allow decryption only inside the processor. But symmetric decryption has non-trivial latency. Adding it to the instruction-fetch critical path is not practical. In the same vein, the keys themselves cannot be stored in memory in plaintext and so, have to be decrypted inside the processor. Asymmetric decryption, however, has orders of magnitude higher latency than symmetric decryption. The main design challenge is thus to absorb or amortize these overheads as much as possible. Furthermore, secure key management being a key component of any encryption scheme, has to be considered at every step of the full-system design. For Polyglot-B, since our TCB is just the processor, we want code and keys to be encrypted at all times while outside the chip. Furthermore, it is preferable that these challenges be addressed with minimal changes in and as simple extensions of already-prevalent system architecture design. In our implementation, modifications mostly occur in the instruction page fault and instruction cache miss pathway.

Instruction Page Fault.

First we explain the basic changes required to enable the simpler version of Polyglot that does not require key-encryption, and then describe simple additions to this framework to support key-encryption.

Since keys are associated with every code page, it is convenient to fetch keys for a particular page while processing the corresponding fault. To do so, we require that last-level PTEs contain the translation as well as the associated key for code pages. The ISR PTE, thus, consists of the actual translation followed by the key for the page. To denote an ISR PTE we use a reserved value in the type field of a page-table value as shown in Figure 3.2.

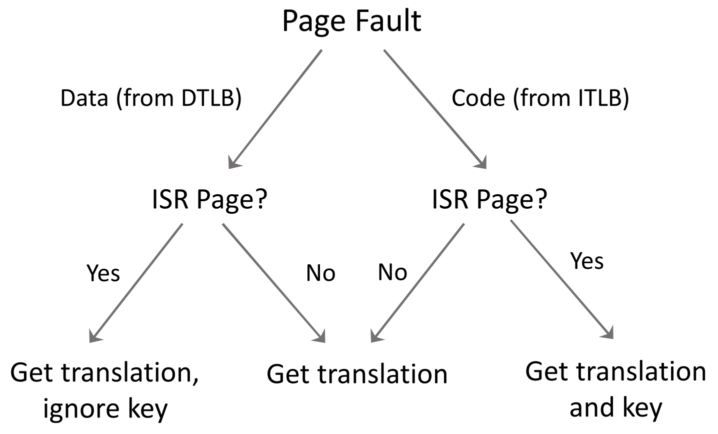


Figure 3.3: ISR page fault handling flowchart.

An interesting design challenge now is that since ISR PTEs are longer than a word and do not otherwise conform to a power-of-two alignment, the traditional method of offset-based PTE fetching does not work. We solved this by extending the original page walk scheme by one more level of indirection. Instead of storing the physical address of another table to offset into, we store the physical address of the ISR PTE entry itself at the penultimate level. This design allows the OS to arrange these entries in whichever way it deems convenient – either as a contiguous table, or as discrete entries in memory. The hardware page-walk scheme is agnostic to this organization.

On a page fault, the origin of the fault (DTLB or ITLB) determines whether a code or data page is expected, and fault-handling is done appropriately (as shown in Figure 3.3). If the fault is on a code page, the table walk proceeds as usual until an entry with the ISR type is encountered. This indicates to the MMU that the next level is an ISR PTE. The walk mechanism procures the key and translation from the physical address found at this level and stored in the ITLB. A normal PTE for a code page fault indicates an unencrypted page. If an invalid entry is found during the walk instead, the page fault handler in the OS is invoked as usual. The OS is then responsible for setting up the tables in a manner expected by the hardware.

On the other hand, if an ISR page is encountered on a data page walk, the key is ignored and only the translation is forwarded to the DTLB. In this way, we effectively allow data and code

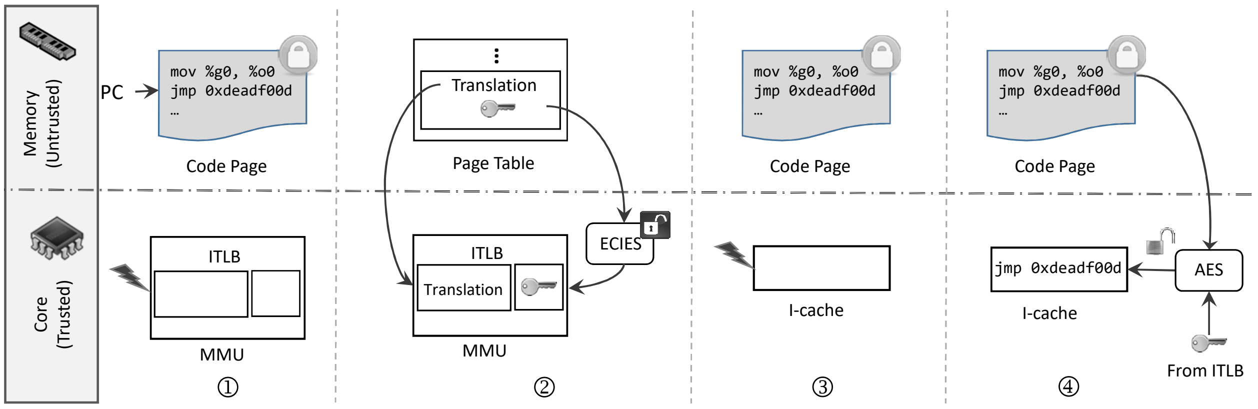


Figure 3.4: Hardware decryption scheme in Polyglot. On a page fault (left half), ISR PTE contents are brought into the ITLB. On a cache miss (right half), a cache line is decrypted using the page key before dropping into I-cache.

to exist in the same page—code accesses to an ISRized page fetch and use the decryption key to decrypt code, while data accesses to the same page will fetch contents as is. This means that code can be treated as data, i.e., can be read and written to. However to allow correct decryption, data and code must either be aligned at cache block width, or, if they exist within the same line, ensure that the data is immutable. This can be done either by the compiler or the user.

An extra level of walk is added to the original mechanism when an ISR PTE is encountered. On encountering it, the hardware uses the the entry as the pointer to the actual translation, and fetches the ISR PTE from that address. The ITLB was modified to hold an additional key-field for each entry. If no ISR entries are found, the walk procedure continues as before.

An overview of how execution proceeds in hardware is shown in Figure 3.4. On encountering a fault to a code page (step ①), the MMU walks the page table and finds the corresponding ISR PTE consisting of the translation and the asymmetrically-encrypted key. At this point, it sequentially fetches the actual translation (step ②) as well as the the key and is deposited into the ITLB. Here onwards, any i-cache miss originating from this page is decrypted with this key (step ④), before being stored in the I-cache. As long as this instruction is not evicted, execution uses the decrypted instruction.

Enabling Key-encryption for Polyglot-B. Doing so requires one additional change in the scheme described above. We require that each processor have an asymmetric key-pair associated with it. While the public portion of it can be revealed, the private portion never leaves the chip.

In this case, the keys in the ISR-PTEs are asymmetrically encrypted with the processor's public key. The only change now would be made in the ISR-PTE fetching step during a page-walk (step ② in the figure). Instead of directly depositing the key in ITLB, it is decrypted first according to the Elliptic Curve Integrated Encrypted Scheme (ECIES) [72]. We ignore the key authentication portion of the standard however.

We added the requisite ECC-162 and SHA-256 accelerators to the MMU, where they perform key decryption. However, we have now inserted asymmetric decryption into the critical path of code-page fault resolution, thus adding a non-trivial latency to it. We rely on the fact that code page faults are not common. We see in our evaluations that this assumption is true for most parts and the performance hit is largely unnoticeable.

Furthermore, since our scheme is easily integrated with page-walking, we make a simple and inexpensive extension in our design to provide encryption at two other, non-page granularities as well. In typical page-tables, the level determines the memory region covered by the PTE. In SPARC32 specifications, a level-3 PTE covers 4kB, level-2 covers 256kB, while level-1 covers 16MB. Our simple optimization, allows a single key to be associated with these memory granularities depending on which level the PTE exists at. This is useful in mapping large regions of code, without using up a lot of space for keys. We found this particularly useful while mapping the kernel's own page tables, especially when set up by the bootloader.

Instruction Cache Miss.

On an I-cache miss, as instructions are fetched from memory, they are decrypted and stored in plaintext in the I-cache. D-cache miss handling remains unaltered. The challenge however is that although symmetric decryption is much faster than asymmetric decryption, instruction fetches are fairly more common. Adding decryption latency to the critical path will cause impractical performance overheads. This is the main reason prior work opt for inexpensive encryption like

XOR or bit-transposition.

To overcome this, we use symmetric encryption in counter mode [40]. In this mode, the actual decryption is performed on a counter value, which is then (traditionally) XOR'ed with the actual data to obtain plaintext. Given the fact that XOR is a cheap operation, if we could determine the appropriate counter value when the miss is encountered and start decrypting early, instruction fetch and decryption could proceed in parallel. Moreover, if fetch latency is greater than decryption latency, no performance overhead would be observable.

The counter itself need not be secret as long as they are not repeated for the same key. We obtain the counter from the lower bits of memory address (see Figure 3.6). Essentially the counter equals the block offset into the page. Notably, since the counter can be determined from the address itself, decryption and fetch request dispatch commence simultaneously. We are able to absorb the decryption latency in this manner.

The Leon3 SPARC32 implementation only has a single level of split caches. Hence we place the AES decryptor at the I-cache/memory interface. In our prototype, since the cache line itself is 32B long while an AES block is 16B, we divide the line into two chunks and decrypt them simultaneously with two separate counter values trivially derivable from the address. A naive way of implementing this process would be to fetch the whole line and decrypt it all at once, thus nullifying two important fetch optimizations. The first of these optimizations streams words into the core as well as the cache as they are fetched instead of delaying the stream until the entire line has arrived. The second optimization starts fetching from the requested portion of the line, which may not necessarily be the first word of it. With counter-mode encryption, we are able to maintain both optimizations. We do so by XOR'ing the decrypted data with the blocks as they stream from memory.

Modern systems, however, have many levels of unified cache. In such systems, we propose that the decryptor be placed between the lowest-level cache (LLC) and memory for energy and performance reasons. However since we only want to decrypt code, we will have to track the source of the miss all the way to this level, and selectively decrypt replies bound for the I-cache. With this scheme instructions are decrypted before being filled into the LLC, and remain decrypted in the lower levels where most of the accesses hit. *Most importantly, since in these systems, memory latency is much higher than symmetric decryption latency, decryption cost can be completely hidden*. Thus symmetric decryption incurs near-zero performance hit, depending on the implementation and operational factors (bandwidth, pipelining, placement on chip, etc.). This is significant especially for Polyglot-A since symmetric decryption latency is the only source of overhead here.

Low-level Shared Caches. Pushing decryption beyond the L1 creates a security vulnerability. Assuming a shared L2, say a cache block was fetched from DRAM into the L2 in response to a data load request. Now if this data is also requested by the L1-I, the block, which came in as data without going through the decrypting process, will be fed as is to the L1-I, thus completely bypassing ISR. An attack vector to exploit this might involve first loading the shell-code, crafted in native ISA, as data, and referencing the same locations for execution soon thereafter while they are still in L2. Conversely, consider the case when a code block in L2 is eventually requested by the L1-D. In such a scenario, a well-crafted attack can read off decrypted instructions just after they are executed.

We prevent this by tracking instruction and data blocks in all shared caches, by adding a bit to each cache block indicating whether it is instruction or data. This bit is set by tracking the source of the miss, i.e., the instruction or the data cache. Cross-sharing between the split caches is then forbidden, either directly or through L2. If L2 receives a request from the L1-I for a block marked data, that block has to be flushed to memory and fetched again, this time going through the decryption process. Similarly, when a block marked instruction is requested as data, it is flushed and fetched again; only now the decryption module will be bypassed.

MMU-less Execution.

The above design changes deal with the common case of an active MMU, when paging is on. Since we want ISR support on at all times, we have to support encrypted execution from the very first instruction executed on boot-up. In this situation, the process is almost the same except the key is acquired not from a PTE but directly from a memory location specified by the user. Design-wise, during this phase, the I-cache changes still remains in play, while the page-walk modifications are disabled.

Portability to Other Architectures.

Although our implementation employs some SPARC-specific optimizations (e.g., page-level based encryption granularity), the fundamental ideas are portable across architectures. Even a fixed-width architecture is not necessary—the same principles apply if an instruction is broken down over two cache lines. However, we do need a way of denoting ISR pages in the PTE[1], as well as an extra tag bit in the I-cache. Most critically, we make or require no changes to the processor core architecture.

Importantly, by completely separating the set of ISR changes within the cache and MMU modules, we keep the core-cache interface intact. Thus, *our mechanism is oblivious to complex core mechanics like super-scalar, branch-prediction, SMT, etc.* We foresee these mechanisms to help our cause by hiding the memory latency associated with code decryption.

As far as software is concerned, the only changes we made that are SPARC-specific is during the booting process of the kernel and bootloader. We see no reason to believe the same should be impossible to port in other architectures.

3.4.3 Design Choice Implications

Encryption Algorithms. We use ECC, instead of RSA (i.e., the more popular asymmetric cipher), since ECC has shorter key lengths and encryption/decryption latencies. Furthermore, the use of counter mode block encryption guarantees that the encrypted code is position dependent and prevents splicing attacks (i.e., copying encrypted code and reusing it elsewhere).

Page Table. Page-level (encryption) granularity implies that brute-forcing, or careful dictionary-like attacks on a particular page, reveals nothing about the rest of the system.

[1]x86 extended page-table (EPT) format has unused mappings [52].

Allowing Data Accesses to ISR Pages. Previously-proposed systems that seek to disallow run-time code-scanning make code pages execute-only. We could easily achieve the same by faulting whenever data page walks encounter an ISR PTE. However, this requires strict segregation between code and data at the page granularity. Although a more secure option, we considered this approach limiting in terms of convenience of development, deployability, and practicality.

Syscall Interface. Our `mmap` variant is used by user applications to map encrypted pages into a process. This is a weak link as it can be exploited by an attacker. Additionally, the original `mmap` is still allowed to load unencrypted code. While the former allowance was indispensable from a practicality standpoint, the latter was necessary for the sake of convenience; variants of our architecture can forbid it. Note that introducing this system call does not make Polyglot more vulnerable to the attack against ASIST that we outlined earlier.

Key Handling. This is one of the most crucial aspects of any crypto-system design. In Polyglot, the symmetric keys are included in the binary. Since these keys are asymmetrically encrypted, Polyglot is safe even against binary code leaks, when an attacker obtains the binary itself or the OS is adversarial[2]. Another implicit assumption is that the private key, specific to a chip, is irretrievable by the attacker. In extreme cases, where the physical tampering of the chip is a concern, the private key could be based on a physically unclonable function [44] that is automatically destroyed if anyone tries to tamper with it.

3.5 Implementation Details

The goal of developing the prototype is to provide a platform for full system bring-up as well as to understand its impacts in the context of fully fleshed out microarchitecture. To this end, we implemented Polyglot in accordance with the design principles described in the previous section. In this section, we provide the specific implementation details of the same.

[2]Note that an adversarial OS can mount additional types of attacks, such as Iago attacks[**iago**].

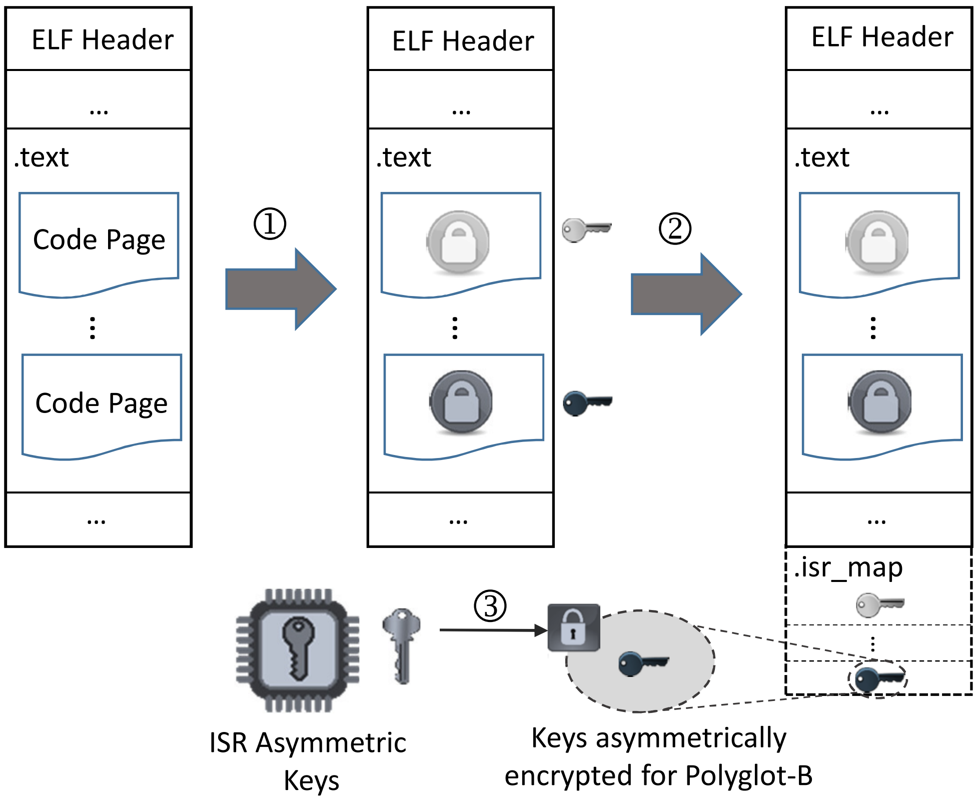


Figure 3.5: ISR binary generation for ELF binaries. Code pages are symmetrically encrypted. Subsequently the symmetric keys are themselves asymmetrically encrypted with the target system's public key, and embedded within a new section of the ELF binary.

3.5.1 Software

Binary-Creation. We modified `objcopy` to perform binary encryption – it takes a modified ELF binary as an input, and encrypts the binary page-wise with AES-128 and adds the key-to-address mappings to the `.isr_map` section as discussed before (Figure 3.5). When key encryption is requested, we use ECC-162 and SHA-256 for encryption according to the ECIES protocol. For our implementation, encryption is automated and we did not have to change the source code of any program to accommodate it. The loader is modified to parse the new ELF section and use the `mmap` variant to communicate the key ranges while loading shared libraries.

Kernel. We had to modify the ELF binary-parsing and virtual memory modules of the kernel to support Polyglot. This involved extracting the key-range mappings from the ELF file, and setting up the page tables for the process appropriately. Other modifications included changing page-fault mechanism, allowing proper page-sharing (in the case of `forks` and shared-libraries), and process tear-down.

Randomizing the kernel itself required two changes. Firstly, we had to change the kernel's own page-mapping to an ISR-PTE version. In SPARC, the level at which a PTE is found in the page-table dictates the memory coverage of that mapping (more details in the next section). Consequentially, the Linux kernel just uses one entry in the first level of the page-table with a coverage of 16MiB. We just changed this to an ISR-PTE of the format described before. Secondly, the SPARC/Linux does some code-patching during boot according to the exact processor version it detects. We modified these portions and hard-coded the proper versions for our hardware, so as to remove all runtime code-patching.

Overall, our modifications added about 1100 LOC to Linux kernel version 3.8.1.

Bootloader. We use `mklinuximg-2.6.36` package [78] as the bootloader. Overall, it boots with the MMU disabled and is responsible for assessing the runtime hardware, copying a portion of itself that persists after the kernel has booted, and then loading the kernel itself. To enable ISR on the bootloader, we encrypt as usual and modify the kernel's and its own page-mappings.

3.5.2 Hardware

We implement our prototype on the 32-bit SPARC-based Leon3 SoC package [66]. Our main hardware changes involves modifying the page-walk and the instruction-miss mechanisms. For the two variants of Polyglot, the only difference is the presence of an asymmetric unit in the page translation fetching pathway.

Page-walk. An extra level of walk is added to the original mechanism when an ISR PTE is encountered. To denote an ISR PTE we use a reserved value in the type field of a page-table value as shown in Figure 3.2. When this is encountered, the hardware uses the rest of the entry as the pointer for the actual translation, and fetches the ISR PTE from that address. The ITLB was modified to hold an additional key-field for each entry. If no ISR entries are found, the walk procedure continues as before. Another important optimization was to modify the memory controller to satisfy ISR requests in burst mode instead of the original configuration of exclusively fetching one word from memory at a time.

For asymmetric decryption, the processor's private key is used in accordance with the ECIES protocol to derive the symmetric key as shown in Figure 3.6, and complements the binary encryption process described before. We added the requisite ECC-162 and SHA-256 accelerators to the MMU, where they perform key decryption in accordance to the protocol. The accelerators themselves were not optimized for this particular operating scenario.

Furthermore, since our scheme is easily integrated with page-walking, we make a simple and inexpensive extension in our design to provide encryption at two other, non-page granularities as well. In typical page-tables, the level determines the memory region covered by the PTE. In SPARC32 specifications, a level-3 PTE covers 4kB, level-2 covers 256kB, while level-1 covers 16MB. Our simple optimization, allows a single key to be associated with these memory granularities depending on which level the PTE exists at. This is useful in mapping large regions of code, without using up a lot of space for keys. We found this particularly useful while mapping the kernel's own page tables, especially when set up by the bootloader.

Instruction-fetch. The Leon3 SPARC32 implementation only has a single level of split caches (L1-I and L1-D). Hence we place the AES decryptor at the I-cache/memory interface. Furthermore, we also did not need to implement the shared cache modifications proposed in the previous section.

For symmetric encryption, we use AES-128 to decrypt the fetched cache-line. In our prototype, since the cache line itself is 32B long, we divide the line into two chunks and decrypt them simultaneously with two separate counter values. A naive way of implementing this process would be to fetch the whole line and decrypt it all at once, thus nullifying two important fetch optimizations. The first of these optimizations streams words into the core as well as the cache as they are fetched instead of delaying the stream until the entire line has arrived. The second optimization

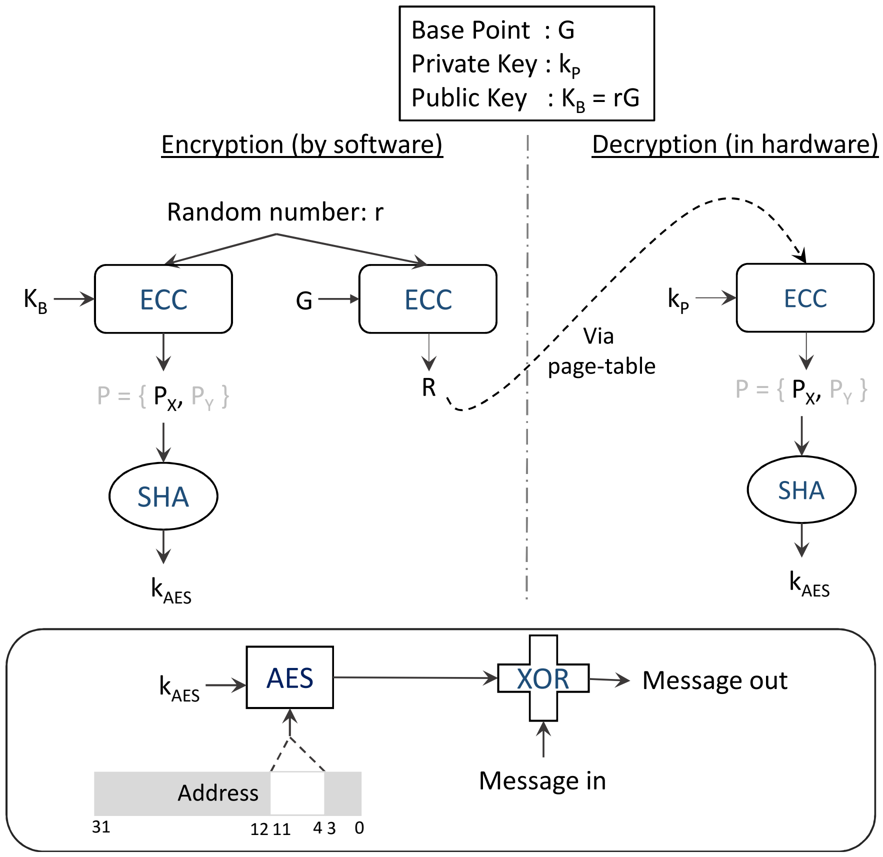


Figure 3.6: Encryption and decryption in the ECIES scheme. Letters in caps indicate points in the Gaussian space which have consist of x- and y- coordinates. Base point, G, is specified by the standards.

starts fetching from the requested portion of the line, which may not necessarily be the first word of it. With counter-mode encryption, we are able to maintain both optimizations.

Since the counter can be determined from the address itself, decryption and fetch request dispatch commence simultaneously. To accommodate the optimizations described above, corresponding block of the encoded cipher is XOR'ed as words are streamed from memory.

Importantly, our modifications to the I-cache involved changing its state machine in a manner that did not affect its interface with the pipeline. Thus, by completely containing the complete set of ISR changes within the cache-MMU modules and keeping transparent to the pipeline, we managed to keep the execution engine in tact.

• **Portability to Other Architectures.** Although our implementation employs some SPARC-specific optimizations (e.g., page-level based encryption granularity), the fundamental ideas are portable across architectures. Even a fixed-width architecture is not necessary – the same principles apply if an instruction is broken down over two cache lines. However, we do need a way of denoting ISR pages in the PTE[3], as well as an extra tag bit in the I-cache. Most critically, we make or require no changes to the processor core architecture.

As far as software is concerned, the only changes we made that are SPARC-specific is during the booting process of the kernel and bootloader. We see no reason to believe the same should be impossible to port in other architectures.

3.6 Security Analysis

Evaluating novel defenses is non-trivial since it changes the basic assumptions for prior attacks thus rendering them ineffective. However, this does not necessarily guarantee its security. Moreover, practical implementations of conceptually secure principles may expose vulnerable interfaces. Here, we examine the rationale behind Polyglot's design decisions and discuss their effectiveness from a security standpoint.

3.6.1 Motivating Strong Encryption

To demonstrate the fact that strong encryption is critically essential, we try to extract the XOR- and transposition- randomization key from a diversified SPARC binary. We use ASIST's encryption parameters as the standard for this analysis.

Our setup is as follows. We use uClibc-0.9.33.2 [118] for SPARC32 as the base binary, which has 1025 functions comprising about 300kB of code. We assume that program bugs allow arbitrary memory reads and that locations of some functions can be leaked through memory disclosures. The plaintext, non-randomized binary is also considered to be available. We emulate three popular randomization schemes – function and basic-block permutation, and instruction replacement/in-

[3]x86 extended page-table (EPT) fomat has unused mappings [52].

sertion. For the latter, we replace/insert instructions aggressively with a probability of 25%. Note that this is not an actual attack, only a feasibility study to quantify the hardness of successfully attacking such systems. We, therefore, purposefully keep our approaches generic by not using any architecture-specific factors and examine the number of functions to be leaked before we can extract the key. Here, we present the fundamental insights for breaking each scheme and the results. The detailed pseudocode of the algorithms used to carry out the following attacks are listed in Appendix 6.1.

Attacking XOR. The main insight we use to break this scheme is that XOR'ing two values, encrypted with the same key, yields the same result as XOR'ing the two corresponding plaintext values: $(a \oplus key) \oplus (b \oplus key) = a \oplus (key \oplus key) \oplus b = a \oplus b$. Additionally, we note that for 128b XOR, every fourth word uses the same key-chunk. By exploiting these points, we needed to examine an average of 5 functions over 100 separately randomized versions of this binary to discover the key.

Attacking Bit-transposition. Even though key-guessing is theoretically easier for bit-transposition than it is for XOR (32! choices for the former as opposed to 2^{128} for the latter), we found that breaking transposition was more complicated in the presence of diversification.

For transposition, correctly identifying the plaintext-ciphertext instructions pair is key, which is hard when code is diversified. The main insight in breaking transposition is that randomization occurs at an instruction granularity. An instruction will, therefore, have the same cipher form regardless of its surrounding instructions or location. Our approach then was to do a frequency analysis of instructions to find possible plaintext-ciphertext pairs. This would have been straightforward had we not used instruction replacement/insertion as one of the randomization schemes. Using the above approach, we needed to examine an average of 51 functions over 100 separately randomized versions of the test binary to guess the key.

Take-aways. Clearly, diversification schemes can be adapted to make these attacks harder. However, any practical attack would certainly be more sophisticated, utilizing not only architectural (e.g., instruction encodings) and/or ABI (e.g., function prologue/epilogue formats) information, but also code-specific knowledge. This makes them significantly more effective. Such attacks can, however, be easily thwarted by employing cryptographically robust encryption. Additionally, if done at a fine granularity, reversing one block does not jeopardize the rest of the system. Polyglot achieves both goals – it uses AES and ECC, and encrypts at the page-granularity – while being performant and scalable.

3.6.2 Effectiveness

Polyglot is essentially meant to demonstrate that ISR is not just a defense against code-injection, but can be an effective counter to code-reuse attacks as well. We discussed its effectiveness in the former avatar in Section 5.2. Here we analyze its effectiveness against the latter.

Firstly, since we rely on static randomization, we are limited by its robustness. However since Polyglot is independent of it, we assume it is impregnable for the sake of this discussion. We also assume that the encryption used is strong enough to be practically irreversible.

As such, static ROP is clearly ineffective since the code memory differs from the attacker's expectation of it based on his own copy. Furthermore, attempts at reading plain code at runtime will also fail since the read code cannot be disassembled sanely. Thus, Polyglot is completely secure against static as well as JIT-ROP attacks based on runtime code-scanning.

However, since ISR does not modify data layout, it is susceptible to information disclosures through data. Consequently, if one were to leak function pointers, it is possible to carry out function-reuse attacks such as ret2libc [35] and COOP [30, 98]. This is because although we randomize the structure of the function, we do not change it functionally. Some forms of data-space randomization [10, 18] and virtual-table protection mechanisms [**bounov16**, **vtrust**] may be effective in this context.

3.6.3 Proof-of-Concept Exploit

The main benefit of ISR against CRAs is that gadget-building attempts that rely on arbitrary memory reading capability to read a process' code at runtime will fail because it reads encrypted code.

Additionally since the code is strongly encrypted it cannot be reversed. Even if the attacker somehow gains that ability, we keep the keys encrypted in memory.

To assess the effectiveness of Polyglot against direct ROP/JOP attacks, we retrofitted CVE-2013-6282 to Linux kernel v3.8.1 (the original vulnerability affected the ARM v6 and v7 platforms only). Next, we ported the respective (publicly-available) exploit [42] to the SPARC architecture and verified that the exploit was successful on the vanilla kernel. Note that the original exploit for CVE-2013-6282 did not use ROP; it used the return-to-user (ret2usr) technique [57], which relies on forcing the kernel to execute shellcode placed in user space. We kept the relevant part(s) for triggering the vulnerability, and replaced the shellcode with a ROP (SPARC) payload. Lastly, we tested the exploit when the same kernel is statically randomized (using a simple scheme that entails function permutation [11, 58] and NOP insertion [57]), and, as expected, it failed, as the respective ROP payload relied on pre-computed gadget addresses, none of which remained correct.

As there are no publicly-available JIT-ROP exploits for the SPARC architecture, we retrofitted an arbitrary read-and-write vulnerability in the `debugfs` pseudo-filesystem [28], reachable by user mode[4]. Next, we modified the previous exploit to abuse this vulnerability and disclose the locations of required gadgets by reading the (randomized) kernel `.text` section. Armed with that information, the payload of the previously-failing exploit is adjusted accordingly. We first tested with ISR enabled, to verify that JIT-ROP works as expected, and indeed bypasses the static randomization scheme(s). Then, we enabled ISR and tried the modified exploit again. This attempt failed, as the code could not be read. In addition, we verified that all code reads yield encrypted content. Finally, reading page tables yielded the asymmetrically encrypted keys. We also verified the above behavior using a hardware debugger by directly reading contents from memory.

[4]The vulnerability allows an attacker to set (from user mode) an `unsigned long` pointer to an arbitrary address in kernel space, and read/write `sizeof(unsigned long)` bytes by dereferencing it.

3.7 Ecosystem

Although we have discussed Polyglot in the context of a local system so far, security is ultimately a full-system property and depends to a large extent on the entire operating ecosystem. Below we will discuss some of the challenges of integrating Polyglot to contemporary software ecosystems, and discuss possible solutions to mitigate them. As we observe below, although these challenges are non-trivial, our scheme allows for a high degree of flexibility and can be adapted to most deployment and distribution environments.

3.7.1 Challenges

Distribution Models. Current models of software distribution are based on dispersing a single binary variant globally, making them simple and efficient from a deployment perspective. On the contrary, Polyglot requires every system have a secret and "personalized" binary version. This requirement for one ISR binary variant per machine is non-trivial to accommodate in current distribution systems for multiple reasons.

Firstly, the process of ISR'izing every binary by randomizing and encrypting has to be carried out either at the vendor or the user side. There are logistical challenges to both. At the vendor side, customizing each binary will incur high overheads and may not be attractive from the perspective of cost- and resource-efficiency for the vendor. At the user side, the main challenges involve error reporting, patching, code signing, and other operations which do not lend themselves well to binary heterogeneity. Impractical overheads (e.g., increased installation and load times) also have to be avoided at all costs. Additionally, in closed source models, user side randomization would entail binary rewriting, which have traditionally had performance and correctness issues [63].

Secondly, since Polyglot's security is fundamentally dependent on the secrecy of the cryptographic keys, any scheme has to prevent their leakage to malicious parties. Vendor side randomization implies that user keys will have to be shared with the vendors. This in turn increases the complexity and attack surface for binary transfering operations, even when the vendors themselves

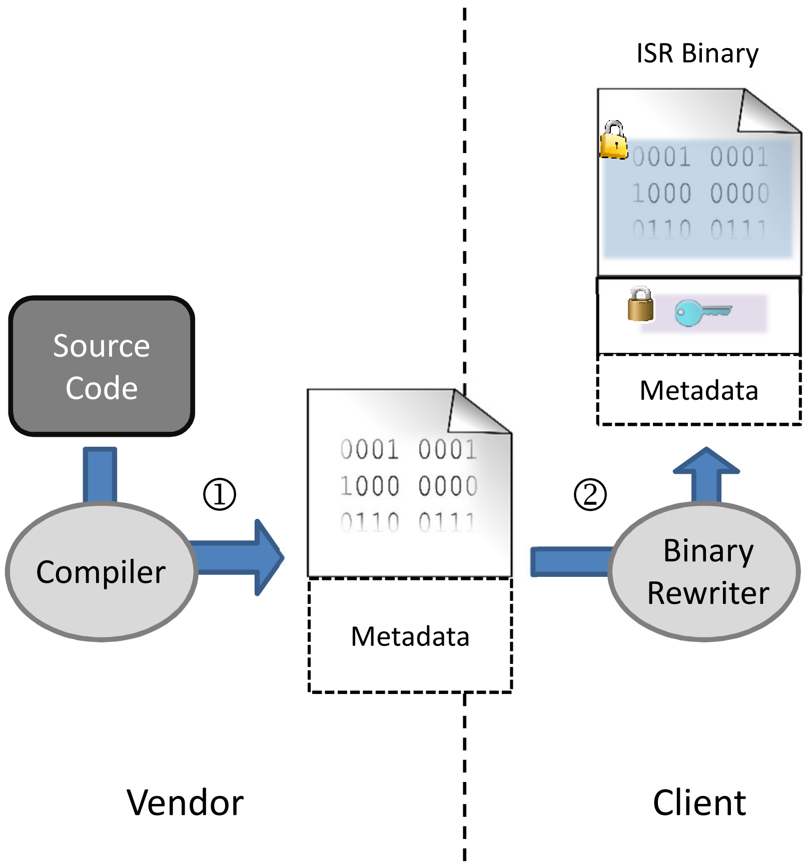


Figure 3.7: Distribution scheme for Polyglot binaries. ① Vendor generates binary and attaches additional metadata to facilitate binary rewriting. ② Binary rewriter at the client side uses this metadata to randomize and encrypt the code, and generates final ISR binary.

are trusted, further disrupting currently established ecosystems.

Key Management. Furthermore, on a more local scope, our proposal so far espouses a single asymmetric key pair per machine. This allows the private key to even be hard-coded into the chip[5]. However, most systems today are multi-user and support virtualization of various guest systems. To have a single key pair in this case would be impractical, undesirable, and insecure.

3.7.2 Distribution Models

To address this challenge, we envision a solution that is largely based on recent work that mitigates the practical problems of integrating code randomization in contemporary distribution systems [61]. This is achieved by generating additional compiler metadata during compilation and linking, with the goal of assisting arbitrary randomization schemes by binary rewriting, as well as code encryption (as shown in Figure 3.7). By maintaining the metadata, this approach also allows binary de-randomization so that bug reporting, patching, and other similar operations can be performed as easily. As we discuss below, this approach is scalable while not being overly disruptive of existing distribution mechanisms.

The open-source model presents the simple case, wherein the source code itself is procured, and subsequently compiled and installed locally[6]. In this case, the binary can be completely ISR'ized within the local machine, thus requiring no modifications in the distribution environment whatsoever. Any vendor side reporting (e.g., for bugs) would be preceded by de-randomization with the help of binary metadata. The closed-source model, on the other hand, is slightly more complicated and cannot accommodate the above solution as naively. In this scenario, we have software vendors distributing a single binary as usual, but packing the compiler metadata into it as well. Once downloaded on the local system, this metadata can then be used for accurate and flexible randomization at the user side. Note that knowledge of the metadata for a publicly available binary does not compromise the randomization of the deployed ISR binaries.

Notably, in both cases, remote distribution, handling, and management of Polyglot keys between the vendor, the requesting machines, and/or any trusted centralized intermediary is unnecessary (although a trust in the developer is a minimum). The encryption process in both cases can be performed within a container or enclave [75] for added security. As such, the additional

[5]In the extreme case, if physical tampering were a concern, the private key could even be based off a physically unclonable function, which are hardware instance-specific signatures, usually based on some physical property unique to it [44].

[6]Modern open source distributions (e.g., Linux-based ones) often employ package management systems, that install pre-compiled binaries rather than compiling individual programs. These systems can use a solution similar to that for closed-source models.

attack surface is reduced to securing the container and its usage locally. The modified installation process, then, would just be similar to the original process, plus invoking this ISR'ization module.

3.7.3 Key Management

Multiple ISR domains within a single machine could be easily accommodated by making the private key programmable, by introducing a special key register that can be loaded with the private key. Isolation techniques, like SGX, can again be employed for safe key setting/unsetting. Thus, in a multi-user system with per-user binaries, session setup could involve loading the user's private key in the key register. Alternatively, a more lax setup could involve loading a key-pair per binary. Ultimately, any point on this spectrum of flexibility can be imposed by a privileged user. Similarly, in a system with virtualization, the hardware can allow each guest to load their own private key. Care must be taken, however, to extend guest isolation to keys (by unloading keys when guests are switched, for instance).

3.8 Evaluation

To evaluate Polyglot, we implemented our design on a Xilinx Virtex5-based XUPV5-LX110T FPGA board. Our implementation is based on the SPARC32-based Leon3 package, and our setup has 256MB of RAM, a portion of which is used as a RAM disk (`ramfs`). On the software side, we used Linux v3.8.1 and uClibc v0.9.33.2. The core utilities were provided through BusyBox v1.23.2. Our system ran with encrypted versions of all the above modules, as well as an encrypted bootloader. For hardware, we used the default Leon3 configuration, sporting an in-order SPARC32 core with no speculation or branch-prediction. Lastly, we use a 64-entry ITLB and a 4-way 32kB I-cache.

It should be noted that even though we run regular workloads on our prototype, the FPGA platform's properties differ from a regular computer in ways that affect our results adversely. Given that TLB and cache misses are the main sources of overhead, overhead reductions are bound to be significant, if structures comparable to those found in contemporary systems are used.[7] Additionally, decryption latency in our case is larger than the latency of memory fetches, while this is not the case for regular systems — our prototype's AES implementation takes 22 cycles to decrypt, while memory fetch, in modern computers, takes about an order of magnitude longer [92].

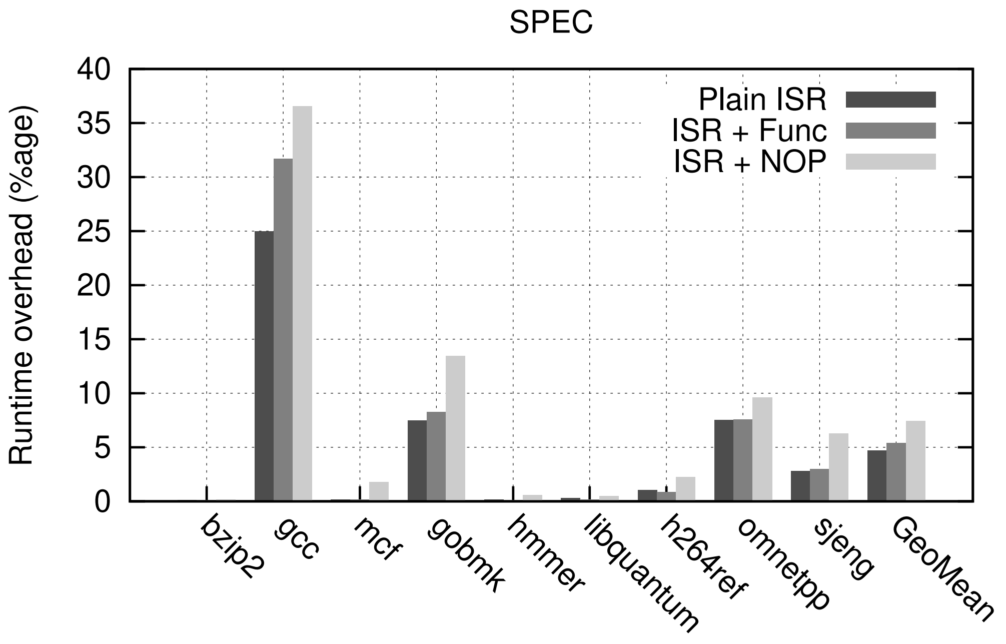
3.8.1 Performance

SPEC. We ran the integer benchmarks of the SPEC CPU2006 benchmark suite [49]. Due to memory limitations on the board, we could only run test inputs and were unable to run all the benchmarks. Note that SPEC benchmarks have been shown to be redundant in metrics (i.e., I-cache and ITLB misses) that are exactly relevant to us [94], and to the extent of our experiments, our results corroborated those findings. Accordingly, `perl` should predictably have similar results as `go`, while `astar` should be similar to `sjeng`. Hence, of the SPECint programs, only `xalanc`'s behavior remains unknown.

Results are presented in Figure 3.8-a. We evaluate Polyglot with three configurations: without randomization, and with function permutation and `NOP` insertion; in the former, the order of functions in the final binary is randomized, while in the latter, we insert up to 8 `NOP`s at function entries and after every call site, preceded by a jump to bypass the `NOP`-sled. Given that we are sensitive to code-misses, these choices are significant (the former does not add to the code size, while the latter does). From the observed data, we see that ISR incurs an overhead of 4.6%. `gcc` performs particularly badly with an overhead of 24.9%, which was a result of an inordinately high rate of ITLB misses (7,376.15 /sec, versus 115.34 /sec for the rest). In fact, if we neglect `gcc`, the rest of the benchmarks have a mean overhead of 2.38%.

For the randomization schemes, function permutation does better, with an overhead of 5.3%, while `NOP` insertion is more expensive: 7.4%. Again, `gcc` was the only outlier with 6% increase in overhead, suffering 15,450 ITLB misses/sec. A similar trend is seen with the `NOP` insertion

[7]For example, ARM Cortex A-15 typically has multi-level caches, and corresponding TLBs, for each level, i.e., a split 32-entry, fully associative L1, and a 512-entry, 4-way unified L2 [4].



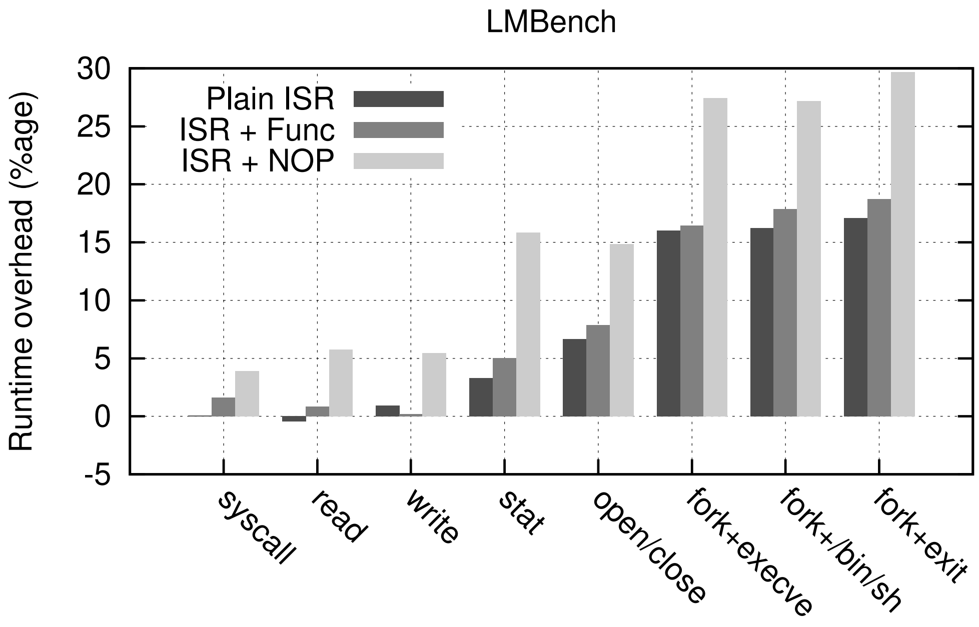


Figure 3.8: Performance overhead for SPEC and LMBench.

scheme. Note though that the `NOP`-randomized binaries themselves exhibit a mean overhead of 1.8%. Based on the above observations, we recommend using in-place code-randomization techniques [89] in Polyglot that do not substantially add to code size. With better code-caching support, however, this might be moot.

Kernel. To evaluate the slowdown caused by the encryption of the kernel, we run the LMBench kernel test suite [76] with the same set of variants. We measure the `null` system call as well as a few other (critical) ones: `read, write, stat, open/close`. Importantly, we also measure process creation latency with `fork+{execve, /bin/sh, exit}`. Figure 3.8-b shows the overhead of encrypted (ISR) over native. We notice similar trends, with overheads for plain ISR being the lowest (0-16%), and ISR+`NOP` exhibiting the highest cost (4-30%).

3.8.2 FPGA Implementation Results

Our modifications to the base Leon3 implementation increased LUT usage from 13,986 to 49,724, a significant portion of which was taken up by the cryptoblocks (approx. 17k LUTs) and the ITLB key storage table (approx. 12k LUTs). Since we did not optimize the accelerators for this particular design, we believe that there is plenty of space for improvement (both in terms of area and performance). For instance, instead of using two separate AES-128 accelerators to decrypt a 256B cache-block, we could merge them to a single accelerator, since they share the same key and have almost identical counters. Furthermore, our modifications to the Leon3 distribution synthesizes at the same clock frequency.

3.9 Related Work

We covered prior ISR work in Section 5.2. In this section, we survey other hardware- and software-based protection schemes, relevant to Polyglot.

• **Code Diversification.** This line of work seeks to prevent CRAs through diversification. More specifically, this flavor of defenses randomize each instance of a binary, or execution, so that the attacker has only a probabilistic chance of succeeding in finding the necessary gadgets. ASLR [91] and many finer-granularity variants of it [63] were proposed towards this end. The common weakness in this class of defenses is that the randomization is static, and relies on the fact that information about a particular instance (of it) cannot be leaked. It has, however, been shown that attacks based on memory disclosure [14, 98, 99, 106] can dynamically harvest gadgets, thereby disproving this assumption.

Recent defenses against the above can be mainly divided into three categories:

① **Execute-only Memory**. Works in this area [5, 17, 29, 46] prevent dynamic code memory scanning by making code pages execute-only. Just this measure, however, is not good enough since code pointers can be harvested from data pages as well [26, 33]. Besides, this class of defenses does not support intermingling data and code.

② **Code-pointer Hiding**. Memory disclosures can be mitigated by preventing the leakage of code pointers, direct (e.g., branch targets) or indirect (i.e., function pointers, return addresses), in the first place. Previous work in this category [6, 29, 30, 70] achieved this via a level of hidden or monitored indirection and/or encoding.

③ **Gadget Invalidation**. Such schemes dynamically modify the program's structure (actively or reactively) so that by the time the (harvested) gadgets are employed they are no longer available [12, 33, 115].

Polyglot broadly falls into category ① . While previous proposals actively disallowed reading code, essentially enforcing a no-read property on code pages, we allow code reads while obfuscating readable code. This, in turn, allows us to intermingle code and data, unlike other proposals. We also show our work to be seamlessly applicable to higher-privileged system software. Lastly, none of the previous schemes offers the degree of protection against static binary leaks that we do.

• **Isolated Execution.** These technologies provide a secure, opaque compartment for programs to execute, without the risk of being spied on by other entities, even those executing at a higher privilege level. First introduced in XOM [116], a slew of software [73, 74] and hardware [41, 65, 112] based techniques have since been proposed; Intel's SGX [75] is an example of the latter category.

In particular, the latter category seeks to provide integrity and confidentiality of both code and data, even in the presence of a malicious operating system. The idea is to achieve security by encrypting code and data outside the TCB (typically the processor), while providing isolation within. Some schemes include memory within their TCB, and simply decrypt the sensitive code at load time, while guaranteeing its integrity thereafter. Others decrypt instructions as they stream into the processor. Although used in other contexts [103, 111, 127], as far as we know, we are the first to employ symmetric encryption in counter mode in order to mask the instruction-decryption overhead. Additionally, isolation techniques cannot cleanly support shared libraries, due to their strict threat model, requiring extensive changes to software. Design changes further need to ensure the proper modularization of secure components lest the attacker gains entry into a compartment. In brief, we avoid the complexities of the larger problem isolation targets, and, thus, are able to provide a more lightweight solution.

3.10 Conclusion

In this chapter, we present the design of Polyglot, a hardware-based ISR scheme, which eschews weak cryptography, as used by previous ISR proposals, by employing AES and ECC at the page granularity. We also develop microarchitectural optimizations to reduce performance overheads typically associated with hardware implementations of these cryptographic algorithms. Our solution enables page sharing between applications and strong encryption with low performance overheads. Furthermore, we allow instructions to be encrypted right from system boot. Most importantly, we show how Polyglot can counter state-of-the-art ROP attacks, which ISR was traditionally considered ineffectual against. These features have not been achieved in any prior ISR implementation, and, therefore, provide a promising primitive.

CHAPTER 4

Practical Memory Safety with REST

Programmers using type unsafe langauges such as C and C++ create many opportunities for attackers to exploit memory safety violations. The severity and prevelance of the memory safety problem combined with the demand for low-overhead solutions has renewed interest in hardware support to mitigate these problems.

In this chapter, we discuss Random Embedded Secret Tokens (*REST*), a simple hardware primitive to provide content-based checks, and show how it can be used to mitigate common types of spatial and temporal memory errors at very low cost. *REST* is simply a very large random value that is embedded into programs. To provide memory safety, *REST* is used to bookend data structures during allocation. If the hardware accesses a *REST* value during execution, due to programming errors or adversarial actions, it reports a privileged memory safety exception.

Implementing *REST* requires 1 bit of metadata per L1 data cache line and a comparator to check for *REST* tokens during a cache fill. The software infrastructure to provide memory safety with *REST* reuses a production-quality memory error detection tool, AddressSanitizer, by changing less than 1.5K lines of code.

REST based memory safety offers several advantages compared to extant methods: (1) it does not require significant redesign of hardware or software, (2) the overhead of heap and stack safety is 2% compared to 40% for a software equivalent (AddressSanitizer), (3) the security of the memory safety implementation is improved compared AddressSanitizer, and (4) *REST* based memory safety can mitigate heap safety errors in legacy binaries without recompilation or source code. These advantages provide a significant step towards continuous runtime memory safety monitoring and mitigation for legacy and new binaries.

4.1 Introduction

Memory corruption errors have been one of the most persistent and long-standing problems in computer security. However, practical and effective solutions to this challenge, although critical to secure program operation, remains an elusive goal to this day. In fact, heap-based memory attacks, exploiting out-of-bounds heap read/writes and use-after-free (UAF) bugs alone, accounted for 80% of root causes that led to remote code execution (RCE) in Microsoft software in 2015 [125].

Previous hardware techniques to address memory safety concerns are broadly based on two approaches — whitelisting safe memory regions and blacklisting (some portion of) unsafe memory regions. Previous work in the former approach, broadly referred to as bounds checking, associates metadata with every pointer indicating the bounds of the data structure it can legitimately access, and flagging any access outside those bounds as memory errors. In the latter approach, commonly called the tripwire approach, critical locations in the address space (for instance, both ends of an array) are marked invalid and any access to them raises a memory violation exception.

Whitelisting approaches [36, 45, 53, 79, 80, 126] offer stronger security guarantees since they monitor all memory accesses against exact bounds. Another advantage to per-pointer metadata is that some of these mechanisms also maintain liveness/version information about data structures they point to, thus detecting dangling pointers in addition to out-of-bound errors. However, they suffer from one or more of the following problems.

① **Performance Overhead.** Since they monitor every pointer dereference, the performance overhead scales with the number of dynamic pointer dereferences. For each of these dereferences there is at least one additional memory instruction for loading the meta data and one comparison operation for checking the data. Even if some overhead can be mitigated by optimizations such as caching, the energy overheads due to the additional instructions are not easily mitigated.

② **Implementation Overhead.** They usually require significant hardware modifications including modifications to the cache hierarchy [36, 79], execution pipeline [36, 53, 79], or even addition of coprocessors [126].

③ **Inaccurate/incomplete Coverage.** Since most of them rely on static pointer analyses for metadata propagation during pointer operations, any inaccuracy in pointer identification leads to incorrect/unstable program behavior. This is especially problematic in the C-memory model, which allows interchangeability between pointer and native data types [24]. Additionally, this also necessitates source code availability, thus preventing such techniques from being compatible with legacy binaries.

Alternatively, tripwires, originally proposed for software, are not a commonly explored technique in hardware [96, 107]. These techniques provide a relatively fast mechanism for marking memory locations invalid. By associating metadata with the locations instead of their pointers, they avoid metadata propagation costs, thus mitigating some drawbacks of whitelisting techniques. However, this comes at the expense of weaker security guarantees since they do not detect all spatial violations (specifically ones that access unmarked regions). In fact, these techniques target a specific access pattern which is commonly responsible for memory overflows. This pattern manifests itself when the program sequentially starts accessing locations beyond the bounds of the data structure (in a loop, for instance). Previous attempts at hardware support for tripwire implementation have required non-trivial hardware modifications (including storage of metadata) and/or incurred non-trivial performance penalty. Furthermore, previous hardware techniques in this category only focus on detecting out-of-bounds accesses and do not address temporal memory safety even though it accounted for 51% of RCE exploits in Microsoft software in 2015, whereas the former accounted for 28.5% [125].

Additionally, checks performed by previous schemes were *tag-based*, in that they use metadata tags, stored in a region separate from program data, to compare and verify access validity. This, in turn, requires (explicit or implicit) out-of-band fetching and processing of metadata.

In this chapter, we propose *Random Embedded Security Tokens* (*REST*), a hardware primitive for content-based checks, and describe a framework based on a primitive enabling programs to blacklist memory regions at a low overhead. This primitive allows the program to store a long unique value, a *token*, in the memory locations to be blacklisted and issues a privileged *REST* exception if it is ever touched with a regular access. We propose a low overhead, low complexity microarchitecture for detecting these tokens. When an L1 data cache line is filled, that memory line is checked for the *REST* token value and if so, marked as such. If a memory instruction accesses that marked line, we throw an exception. These hardware modifications are trivial, requiring no modifications to either the core design, or the coherence and consistency implementations of the cache, even for multicore, out-of-order processors. Ours is also the first scheme to rely on *content-based* checks wherein the metadata is stored alongside program data and requires no modification of the program's overall memory layout. Token checks are performed directly on all data accessed by the program and requires no behind-the-scene metadata processing.

The rest of our framework is based on a software tripwire-based scheme, AddressSanitizer (ASan) [100], which consists of a compilation framework and runtime library that automatically fortifies programs against memory errors without any programmer effort. ASan is a highly popular memory error detector, used in the testing infrastructure of production softwares such as Firefox [43] and Chromium [25]. However, due to its high performance overhead (~1.4x), it is mainly used for software testing and debugging, not in deployment builds. Comparatively, *REST* incurs an overhead of 2% on the SPEC benchmarks while not only providing the same scope of protection as ASan, but even improving its security in several aspects. Moreover, our technique is also able to provide heap safety for legacy binaries at similar overheads. Additionally, as we show later, the observed overheads are completely attributable to the software framework; our hardware

```
1  int tls1_process_heartbeat(SSL *s) {
2    unsigned char *p = &s->s3->rrec.data[0];
3    unsigned short hbtype = *p++;
4    unsigned int payload;
5
6    /* Attacker-controlled memcpy length */
7    n2s(p, payload);
8
9    if (hbtype == TLS1_HB_REQUEST) {
10     unsigned char *buffer =
11                 OPENSSL_malloc(payload);
12
13     /* Vulnerable OOB memory read */
14     memcpy(buffer, p, payload);
15     ...
```

Listing 4.1: Heartbleed out-of-bounds memory read bug.

primitive incurs nearly zero additional performance overhead, and has negligible implementation complexity.

We illustrate the basic idea of our defense with a simplified version of CVE-2014-0160 [31], a bug commonly known as the Heartbleed vulnerability reported in OpenSSL 1.0.1, as shown in the code shown in Listing 4.1.

Line 7 in the listed routine contains the overflow bug wherein the payload length, `payload`, is used to determine the size of data to be copied into the response packet without checking its validity. The resulting exploit can then be used to leak sensitive information such as passwords, usernames, secret keys etc., to the client. Furthermore, common protections involving (stack or heap) canaries would be unable to detect this attack, since it involves a read overflow and does not otherwise corrupt any program state. To prevent this, *REST* tokens are placed around the source buffer to be copied, so that when access goes beyond its bounds, a security exception is triggered, as shown in Figure 4.1.

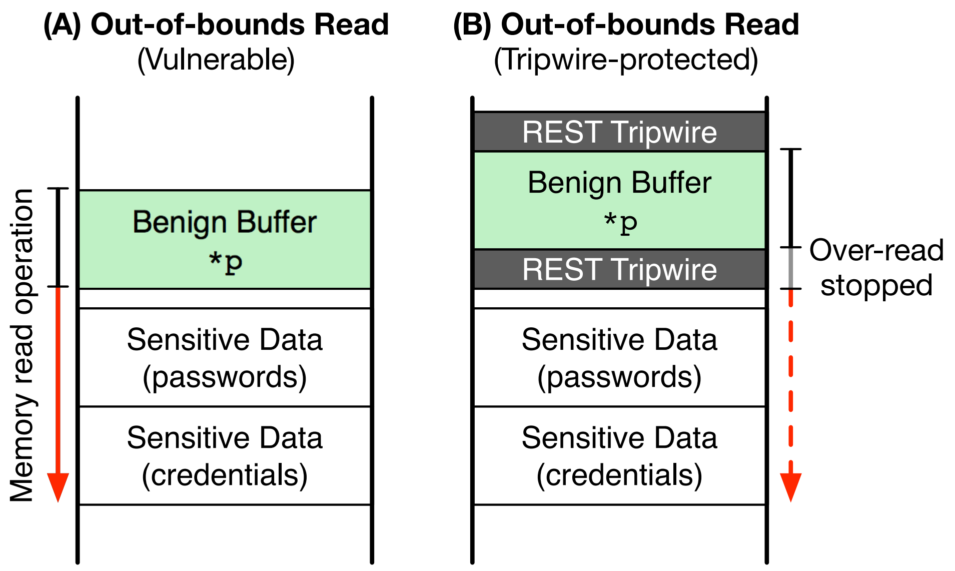


Figure 4.1: (A) Unsanitized `memcpy` bug reads sensitive data outside the benign buffer. (B) *REST* tokens placed around the buffer detects this out-of-bounds access.

4.2 Motivation

Functionally, *REST* provides similar safety features as ASan, a state-of-the-art memory error detector widely used for verification and debugging. Despite its effectiveness, it is not used as a live security scheme due to its performance overheads.

ASan implements a software tripwire-based system, wherein blacklisted zones (also called *redzones*) are placed around sensitive data structures. It then detects erroneous program behavior that leads to illegitimate accesses of these location (in case of an overflow, for instance). To do so, ASan primarily relies on two techniques — shadow memory and memory access instrumentation (see Figure 4.2). Firstly, it reserves a chunk of memory, called *shadow memory*, that contains metadata and should never be explicitly accessed by the program. The rest of the address space maps to its corresponding shadow location via a simple mapping function. Additionally, ASan imposes memory-safe program behavior by checking the validity of every memory access against the metadata for the accessed location. This is achieved by statically instrumenting the program to insert checks before every memory access. When data structures are deallocated, the corresponding

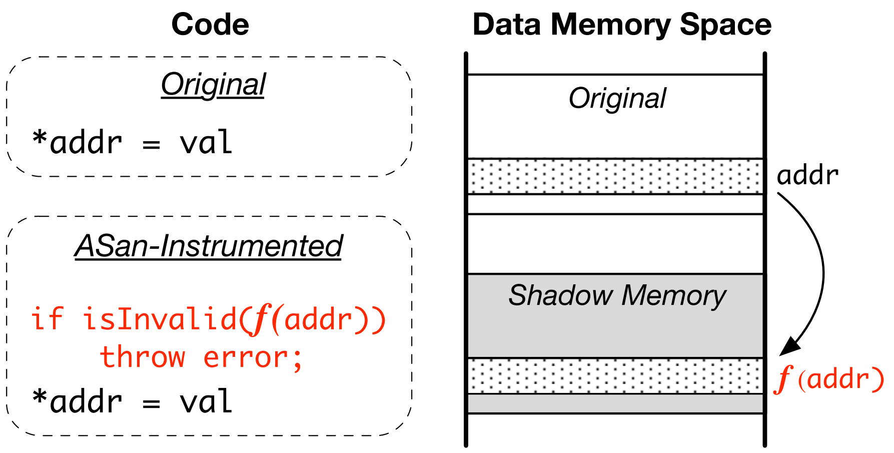


Figure 4.2: Code and address space transformation done by ASan. Memory accesses are instrumented to check against the corresponding value in the shadow memory (dark region in figure), calculable with a simple mapping function, f.

regions are marked invalid by zeroing out the corresponding metadata.

Sources of Overhead. In terms of performance, ASan has four major sources of overhead. ① ASan uses a *custom allocator* designed with security in mind that maintains separate pools for free memory (from which new allocations are made) and deallocated memory (consisting of recent deallocations), and allows virtually no allocation reuse in order to prevent use-after-free (UAF) errors. Hence, it is slower than other allocators which are primarily designed with performance as a first-order feature. ② ASan inserts code at function prologues and epilogues to *modify the stack frame* by inserting and aligning stack variables in order to deter stack attacks. ③ *Instrumentation* for validating memory accesses, as discussed above, also contributes towards ASan's slowdown. ④ Furthermore, since memory checks cannot be inserted in third party libraries, ASan partially mitigates the problem by *intercepting common* `libc` *data-handling API calls* (e.g.,`strcpy` and `memcpy`) to verify that no invalid access occurs therein for the particular set of arguments.

Figure 4.3 provides a breakdown of these components for the SPEC CPU2006 benchmarks simulated on an in-order core[1]. As we see in the figure, memory access checks (③ and ④) account for the most persistent and grievous source of overhead, although the allocator also contributes

[1]The memory side configuration is same as in Table 4.2.

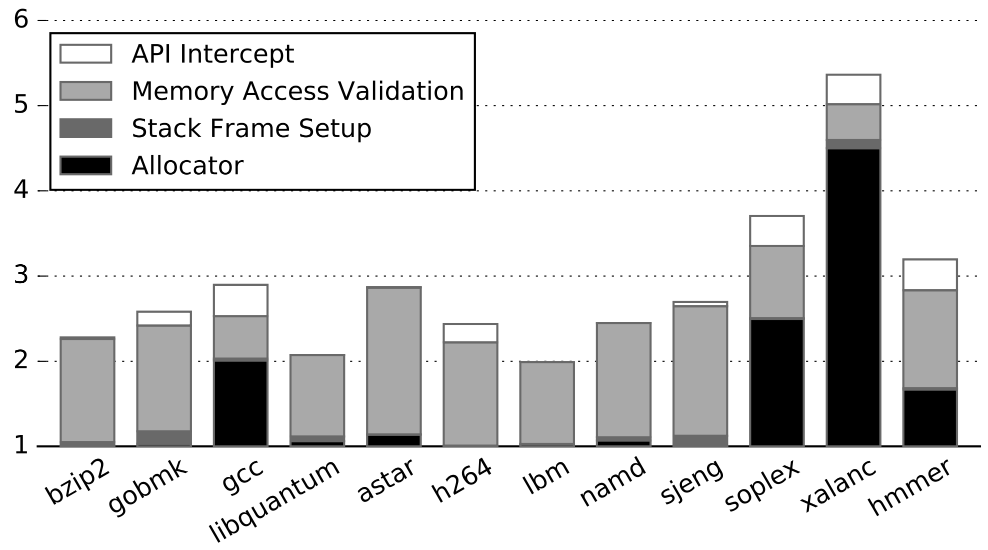


Figure 4.3: Breakdown of various sources of overhead in ASan with respect to a plain binary using `libc`'s allocator.

significantly for benchmarks that make frequent heap allocations. In the subsequent sections, we show how our scheme removes the overheads associated with most of these components.

Notably, ASan's developers also consider potential hardware assistance [2] to speed up metadata lookup and memory access checks transparently by encoding the corresponding logic within a single architectural instruction in a design similar to Watchdoglite [80]. As such, ASan-fortified programs could compress the entire memory-access validation into a single instruction, thus optimizing the expensive operations, but not necessarily removing them. Furthermore, although Watchdoglite has been shown to be highly effective for memory safety in its own respect, such a design would suffer from some of the drawbacks of bounds checking schemes discussed earlier and would necessarily require recompilation. We discuss and contrast similar hardware techniques in more detail in section 4.7.

4.3 Hardware Design

Since *REST* hardware aims to detect and flag accesses to tokens, our main challenge is to be performant by hiding latencies associated with additional memory checks, while maintaining existing microarchitectural optimizations and ensuring the integrity of token semantics. Modifications for *REST* consists of extending the ISA with two new instructions and an exception type, as well as microarchitectural modifications to support them with minimal overhead. We discuss these aspects of the *REST* primitive design below.

4.3.1 ISA Modifications

The width of the token is that of a cache line (64B in our system), and its value is held in a token configuration register (which is not directly accessible to user-level applications). Two instructions are added to set (store) and unset (remove) tokens in the application:

① `arm <reg>` This instruction stores a token at location specified in register `reg`, which should be capable of addressing the entire address space. The implicit operand in this instruction is the token value stored in the token configuration register. The specified location has to be aligned to the token width, otherwise a precise invalid *REST* instruction exception is generated.

② `disarm <reg>` This instruction overwrites a token at location specified in the register `<reg>`, which should be capable of addressing the entire address space, with the value zero. The specified location also has to be aligned to the token width, otherwise a precise invalid *REST* instruction exception is generated. Additionally, in case there is no token at the location, a *REST* exception is generated as well.

When a *REST* exception is triggered, the exception is handled by the next higher privilege level. If the exception is generated at the highest privilege mode, we consider it a fatal exception. We also assume the faulting address is passed in an existing register.

Setting the token value is done through a store instruction that writes to a memory-mapped address. Depending on the token width, one or more stores might be necessary to set the full token value. This operation can only be performed by a higher privileged mode.

We also provide two modes of operation, *debug* and *secure*. The secure mode is expected to be the typical mode of operation for programs in deployment and does not guarantee precise recovery of program state on a *REST* exception (behavior for other exceptions remains unchanged). In the

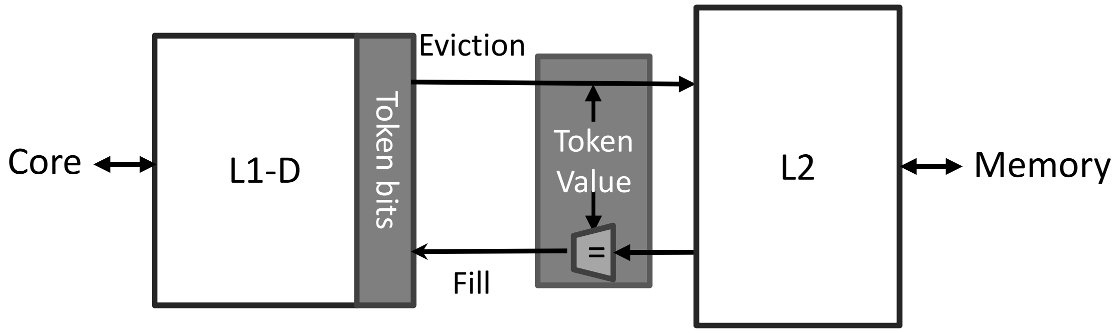


Figure 4.4: Hardware modifications for *REST* include an extra metadata bit per cache line in L1 data cache indicating whether it contains a token, and the token detector to examine incoming data from lower caches and fill the token value into evicted lines.

debug mode, the entire program state at the time of *REST* exception can be precisely recovered by the exception handler. Thus, this mode is intended for use by developers. The current mode of operation can be configured by setting a bit in the token configuration register.

4.3.2 Microarchitecture

In our design, loads and stores check the accessed data against the token value and raise an exception in case of a match. Thus, logically each load becomes a load followed by a comparison of the loaded value with the token, while a store becomes a load of the value to be overwritten, a comparison with the token value, followed by the store. Additionally, reading and/or writing a 64B token value would involve data transfers over multiple cycles, since data buses are narrower. Naively implemented, this could increase the latency and energy of memory operations significantly.

We show a novel construction for *REST* that minimizes changes to load store pipelines and latency for memory operations. Our key observation is that checks necessary for the *REST* system can be performed when the cache lines are installed or accessed instead of explicitly fetching the values and checking them.

Cache Modifications. We extend each cache line in the L1 data cache to include one additional bit to indicate if that line contains a token. Note that since tokens are aligned, a token is guaranteed to be contained within a single line. When a cache line is being installed, the value of that line is compared to the token value register and in case of a match, the token bit corresponding to that line is set. Since cache fills typically happen over multiple cycles the token comparison can be decomposed into small manageable compare operation, say a 32b compare per cache fill stage, to reduce energy. After the fill, memory operations that access lines with the token bit set are flagged to throw a *REST* exception.

A disarm instruction unsets the token bit corresponding to the accessed line and concurrently zeroes out the entire cache line. Since such an operation involves all data banks of the cache, disarm writes incur an additional, typically one cycle, latency. Additionally, disarms raise a *REST* exception if the token bit is not set on the destination line, thus ensuring that the program can only disarm armed locations. The arm instruction sets the token bit of the accessed line, but does not write the token value into it; the token values are written out when the line is evicted from the L1 data cache. This construction ensures that arm operations that hit in the cache complete in a single cycle, despite being a wide write. Our construction works naturally for write-allocate caches, which is one of the most commonly used allocation policies supported in current microarchitectures.

LSQ Modification. Since arm and disarm instructions write values, they are functionally stores and handled as such in the microarchitecture with one key difference. Unlike stores, the arm and disarm instructions should not forward their values to younger loads, as this will violate the invariant that the *REST* token must be a secret. One simple way to provide this invariant is to serialize the execution of arm and disarm execution, i.e., ensure that an arm or disarm instruction is the only inflight instruction when it is encountered in the decode stage. This option, while simple to implement, can introduce significant performance penalities.

Instead of serialization, we next describe design to prevent such forwarding in a common (and complex) structure used to support store to load forwarding, the load-store queue (LSQ). Consider a scenario where an arm request is closely followed by a read to the same cache line. In this case the load may “hit" the in-flight arm in the LSQ, thus forwarding an otherwise illegal read. When

Action	LSQ	Cache Hit	Cache Miss
Arm	Create entry in SQ, tag as arm.	Set token bit	Fetch line, set token bit.
Disarm	Raise exception if SQ has disarm for same location. Else insert entry with no store value in SQ, tag as disarm.	If token bit unset, raise exception. Else clear line, unset token bit(s).	Fetch line, set token bit if it has token. Proceed as hit.
Load	If value can be forwarded from armed SQ entry, raise exception. As usual otherwise.	If token bit set, raise exception. Else read data.	Fetch line, set token bit if it has token. Proceed as hit.
Store (Secure)	Raise exception if SQ has arm for same location. As usual otherwise.	If token bit set, raise exception. Else write data.	Fetch line, set token bit if it has token. Proceed as hit.
Store (Debug)	Raise exception if SQ has arm for same location. As usual otherwise.	If token bit set, raise exception. Else write data.	Fetch line, set token bit if it has token. Delay store commit till ack from L1-D.
Coherence Msgs.	N/A	As usual.	As usual.
Eviction	N/A	If token bit set, fill token value in outgoing packet.	N/A

Table 4.1: Actions taken on various operations for L1-D cache hits and misses.

this case is encountered, we throw a privileged *REST* exception.

This exception support can be implemented without any additional state or impact on LSQ access timing. To do so, we incorporate the *REST* violation check into the existing matching logic simply by breaking the match down to perform two matches — one an address match for the cache line address and another for the remaining — and adding a few logic gates (as shown in Figure 4.5). Additionally since the arm and disarm write values are implicit and known by the cache, we do not attach a value with the corresponding entry in the store queue. With these modifications, LSQ access latencies and data widths remain unchanged despite the introduction of very wide writes. Such address modifications may be necessary at other places in the microarchitecture where store to load forwarding may occur.

Exception Reporting. We can further optimize the performance cost of *REST* by being flexble about how and when exceptions are reported. Supporting precise exceptions with *REST* requires disabling performance optimizations such as critical-word first, and early and eager commit of stores that are common in modern processors. However, *REST* exceptions do not have to be re-

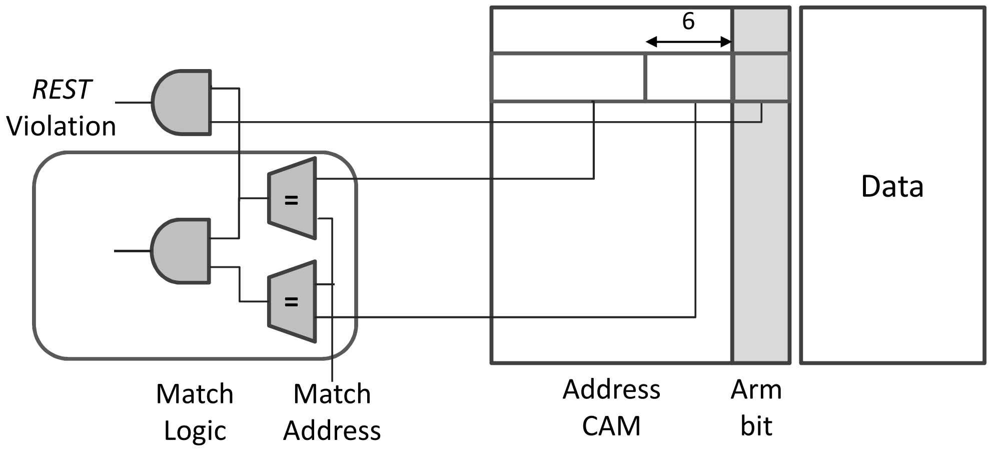


Figure 4.5: Modifications to the LSQ. Added structures are noted in darker shade.

ported precisely especially when it is used for monitoring for security violations during deployment as in these cases the user is typically interested in knowing if a security violation occurred or not, and not the state of the machine when the violation occurred.

If the L1 data cache supports critical-word first fetching, the access request may be satisfied before the whole line has arrived and a match determined. This creates the possibility of a delay between load commit and the security check, especially when the load is at the head of the ROB and is committed as soon as the critical word arrives but the entire line has not. In the debug mode, loads are not released from the miss status handling registers (MSHRs) as long as the delivered word partially matches the token value. On a mismatch, the load is released without any performance penalty. In the secure mode, *REST* exception is reported independent of the load commit.

Additionally, since stores are committed from the ROB as soon as the store/arm/disarm becomes the oldest instruction, *REST* violations due to a faulty access might not be resolved in time. By the time the violation is detected at the cache and the response is received at the ROB, the offending instruction may have retired. This will result in an imprecise *REST* exception. In the debug mode, we guarantee precise exceptions by delaying store commit until writes complete. These modifications in commit logic are summarized in Figure 4.6.

Modifying Token Width. Although the width of our token is same as that of a cache line,

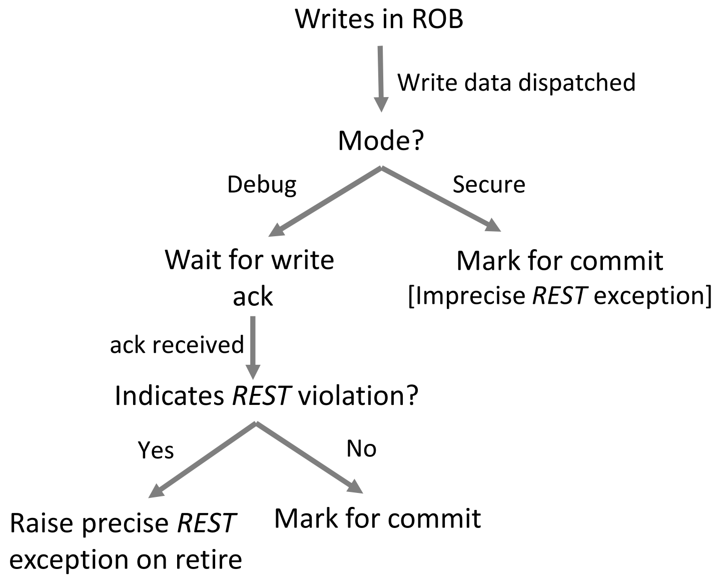

Writes in ROB
Write data dispatched
Mode?
Debug
Secure
Wait for write ack
Mark for commit [Imprecise *REST* exception]
ack received
Indicates *REST* violation?
Yes
No
Raise precise *REST* exception on retire
Mark for commit

Figure 4.6: Flowchart showing write commit logic for *REST*.

it can be reduced for security and performance reasons. For instance, instead of a full cache line width, half or quarter cache line tokens may be used. Most changes described above can be simply scaled to accommodate this. For instance, the token value register can be smaller, and the number of token bits per line will increase to 2 and 4 for 32- and 16-byte tokens respectively. Because of the simplicity of scaling, the same system can accommodate multiple token widths and switch according to the needs of the executing program.

4.4 Software Design

The *REST* primitive described above provides programs the capability to blacklist certain memory locations and disallow regular accesses to them. In this section, we describe how programs can leverage this primitive to obtain spatial and temporal memory safety with little to no changes in its construction and/or layout.

4.4.1 Userlevel Support

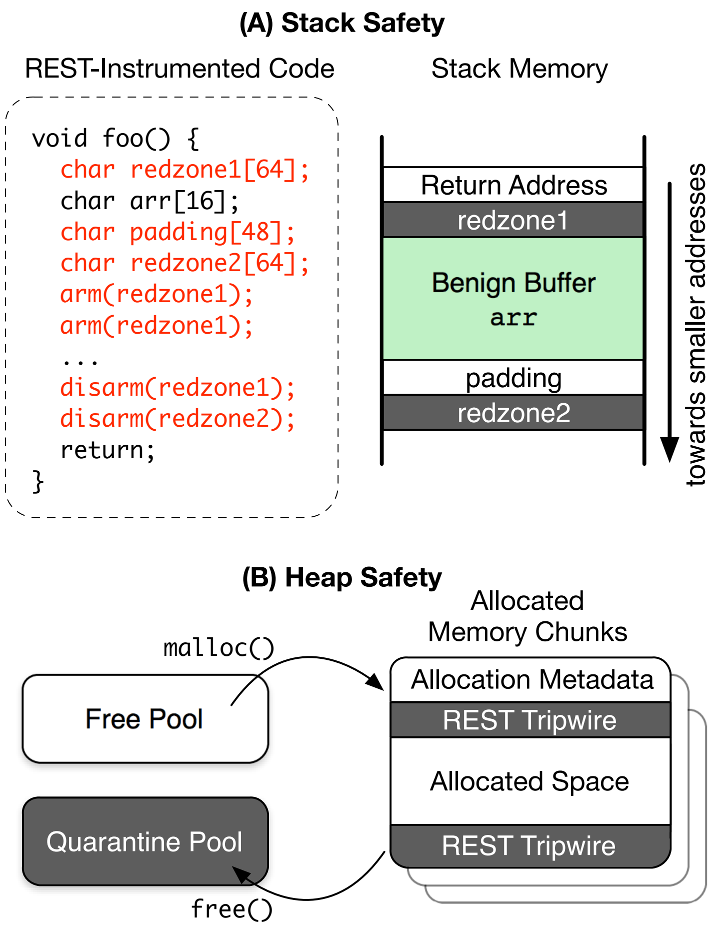
We base our software design on ASan, which is a highly popular open-source memory error detection tool. *REST*'s software framework, however, uses tokens instead of metadata to denote redzones. This obviates two major components of ASan's original design. Since our hardware continuously detects access to tokens without software intervention, monitoring every program read and write in software becomes unnecessary. Thus, memory operations no longer need to be instrumented for checking access validity. Secondly, since *REST* tokens do not require separate maintenance of metadata, the need for shadow memory is eliminated as well. Combined, this essentially eliminates the two major sources of ASan's performance and memory overheads, simplifying its implementation complexity.

Protecting the Stack. As shown in Figure 4.7, protecting vulnerable stack variables involves placing redzones around it. This is done by code added at the function prologue, so redzones isolate these variables from the other local variables. The size of each redzone is chosen as a multiple of the token width and is based on the size of the data structure. Subsequently, overflows during the frame's lifetime are detected when accesses go past their boundaries and into one of the redzones. Code is also inserted at the function epilogue to clean up the tokens so that future frames inherit a clean stack.

Since the above changes involve modifying the stack layout, *REST* requires that binaries be compiled with our plugin. However, since stack attacks have become an insignificant threat vector in recent years [125], users may also choose to forego stack protection, if performance is a concern, and just opt for heap protection as described next.

Protecting the Heap. *REST* secures the heap with a custom allocator adapted from ASan. Spatial heap protection is provided by ensuring that the allocator surrounds every allocation with redzones (see Figure 4.7). These redzones not only separate the allocations from each other but also from the metadata.

Temporal bugs are prevented by filling all freed allocations with tokens and placing them in a separate *quarantine pool*, instead of the pool of free memory from which new allocations are



```
void foo() {
  char redzone1[64];
  char arr[16];
  char padding[48];
  char redzone2[64];
  arm(redzone1);
  arm(redzone1);
  ...
  disarm(redzone1);
  disarm(redzone2);
  return;
}
```

Figure 4.7: (A) For stack safety we instrument the program to insert tokens around vulnerable buffers. (B) Our allocator provides heap safety by surrounding allocations with tokens and blacklisting deallocated regions in the quarantine pool.

assigned. They remain there until the free memory pool has been sufficiently consumed at which point, they are disarmed and released for reallocation. Hence, UAF attacks are mitigated since freed allocations remain blacklisted and any attempts at accessing them via dangling pointers or double frees are caught.

We make one modification to ASan's free pool management however. ASan originally maintains the invariant that all entries in its free and quarantine pool be blacklisted. This necessitates blacklisting newly mapped region from the system, and mark them valid just before allocation. For *REST* we relax the invariant to guarantee that only quarantined regions are blacklisted while those in the free pool are zeroed. This is because blacklisting, in our case, involves storing tokens all over the newly mapped regions and is hence slower than just rewriting corresponding metadata as is done by ASan. Our invariant is maintained for reused regions since disarms zero out memory before they are moved to the free pool and reallocated, thus avoiding uninitialized data leaks.

One key advantage of our protection mechanism is that it works with legacy binaries. Since *REST* performs memory access checks in hardware, heap protection in our case does not require any instrumentation of the original program and can thus be availed even by legacy binaries, as long as our custom allocator is used (with `LD_PRELOAD` environment variable in Unix-based systems, for instance).

Porting to `dlmalloc`. ASan was originally designed for effectiveness, not performance. Its allocator follows the same principle and is, hence, significantly slower than other popular allocators. So, in order to test the portability of our scheme and simultaneously switch to a lower overhead alternative, we implemented the principles outlined above to `dlmalloc`, a classic allocator from which `glibc`'s allocator is derived[2]. Functionally, the modifications were exactly as described above. The quarantine list was implemented as a simple queue of freed allocations. Once the ceiling of quarantined memory is reached, allocations are released in FIFO order until the size of the pool goes below the stipulated maximum. This naive releasing scheme, however, introduces the security problem of predictability. We discuss this aspect more in Section 4.5.

Overall, the entire porting process was fairly straight-forward and did not require modification of the allocator's core algorithm. Overall, our changes added only about 200 LoC to the original.

4.4.2 System Level Support

At the system level, we propose having a single token value. As will be discussed in section 4.5, the token widths are sufficiently long that the chances of a random program value matching a token is vanishingly small (see subsection 4.5.2). However, leaking this value via physical or side-channel attacks might still be possible and would compromise the entire system. So periodically this token value can be rotated (at reboot, for instance). Our design for heap safety allows this model without

[2]Besides other optimizations, `dlmalloc` does not have thread safety, whereas the `glibc` allocator does.

the need for recompilation.

Alternatively, a unique token value could be used for every process with the OS maintaining them across context switches. This design requires some changes to the OS such as the generation of token values and the ability to deal with tokens from different processes when processes are cloned or communicate with each other.

4.5 Hardware/Software Security

4.5.1 Threat Model

In line with recent related work regarding memory error based attacks and defenses, we assume the following in and of our system. The target program has one or more memory vulnerabilities, that can be exploited by an attacker operating at the same privilege level to gain arbitrary read and/or write capabilities within the execution context. We do not make any assumptions as to how these vulnerabilities arise or what attack vectors are used to exploit them. We also assume that the target has common hardware defenses available in most systems today (e.g., NX-bit). Furthermore, we assume that the hardware is trusted and does not contain and/or is not subjected to bugs arising from improper usage parameters resulting in glitching, physical, or side-channel attacks.

4.5.2 Hardware Discussion

In this section, we discuss the security implications of our token primitive independent of the software framework utilizing it.

• **Token Width.** A key assumption of our design is that token detection does not suffer from false positives, which occur when token exceptions are triggered by a legitimate chunk of program data. Three conditions have to be met for this.

① The data chunk equals token value,

② It is aligned to token width, and

③ It is fetched into the L1 data cache, thus passing through the token detector. If data transiently acquires the token value while already in L1 data cache or any other part of the memory subsystem, no exception is raised.

To avoid false positives, it is therefore critical not only to choose a properly random token value but also an appropriate token width. In our design we choose a width of 512 bits, which makes the chances for a program data chunk causing a false positive less than $\frac{1}{2^{512}}$[3]. Alternatively, smaller token widths of 256, 128 or even 64 bits are also usable for overhead reduction. As discussed in section 4.3, these widths should entail minimal changes in our original design and can be supported simultaneously. In such systems, depending on the requirement models, it is possible to have a scheme where programs execute with low token widths first and restart with higher token widths in case a *REST* exception is detected.

• **Immutability and Unmaskability.** *REST* makes sure that once a token is set, it can only be removed through a disarm operation and cannot be otherwise overwritten (or even read) by any process at the current privilege level. Additionally, *REST* exceptions cannot be masked from the same privilege level. These measures ensure that adversaries cannot exploit inter-process, inter-core, or inter-cache interactions to bypass token semantics.

• **Detector Placement.** We place our detector at the the L1 data cache in order to keep the other caches unmodified and hence, minimize design costs. Consequently, however, *REST* does not catch token accesses via means that completely sidestep the cache (e.g., DMA).

4.5.3 Software Discussion

While *REST* is based on ASan, it improves upon ASan's security in a number of ways. In this section, we elaborate upon the weaknesses of ASan, if/how *REST* mitigates them, and whether we

[3]For simple reference, a maximum of 2^{48} token-aligned data chunks can reside in a 64b address space simultaneously. Additionally, a modern system operating at 3GHz would need ~10^{145} years to guess a 512b random value via simple increment operations.

introduce any vulnerabilities of our own.

• **False Negatives.** Token width affects token alignment and therefore, the target data structures[4]. Imposing this granularity on program data, in turn, introduces small gaps between variables. For instance, in Figure 4.7, *REST* adds a pad space adjacent to an array to conform to the granularity requirement (64B in the figure). This introduces the scope for false negatives, wherein *REST* is unable to detect overflows that are small enough to spill into the pad, but not into the token itself. This implies that although we still protect against read/write overflows, our system is vulnerable uninitialized data leaks in the stack [69], which can be simply prevented by zeroing out the padding or using narrower tokens. Uninitialized data leaks are not a problem in the heap, however, due to our invariant that all regions in our allocator's free pool are zeroed.

• **Brute-force Disarm.** Our decision to mandate precise specification of an armed location while disarming is to counter a scenario when an attacker has somehow obtained control of a disarm gadget (i.e., can influence its address argument), but does not accurately know the layout regarding which memory locations are specifically armed. In such a scenario, this design decision prevents attackers from blindly disarming swathes of memory regions. Properly compiled code, however, should have no problems due to this stipulation.

• **Privilege.** Although used in some security mechanisms [121], ASan was primarily developed for debugging. While it can serve as a security tool under weak threat models and performance requirements, realistically it has limited utility as one. This is primarily because its framework is implemented at the same privilege level as the program itself. While the location of shadow memory is randomized, it remains open to memory disclosure attacks, upon which the metadata can be easily tampered with. Memory access monitoring, while statically baked into the program, can also be subverted with carefully crafted code gadgets or even simple code injection. We overcome this issue by raising a *REST* violation on a token, regardless of privilege.

• **Handling `setjmp/longjmp`.** Since the program can neither probe for the presence of a token, nor does it keep a log of all armed locations, disarming necessarily needs to be carried out

[4]ASan also imposes alignment on protected data structures [1].

in the presence of a known reference point. For the stack, frames serve this purpose, i.e., for a given function, arms/disarms occur at fixed offsets within the frame. Consequently, we could not extend *REST*'s protection to support programs that use `setjmp/longjmp` since these instructions alter the stack layout. ASan takes a very conservative approach in such cases by zeroing out the metadata, and hence whitelisting the entire region of the current stack. We cannot take the same approach since we do not keep track of active tokens on the stack. Providing a secure and cheap mechanism for handling this case remains a topic of future research.

• **Predictability.** Our design, as well as ASan's, suffers from predictable layout as attackers can simply jump over redzones (countered to some extent by adjusting redzone size according to the buffer size). Although we do not use it in our system, we recommend that *REST* be used in conjunction with some variant of layout randomization, depending on the usage scenario. Layout randomization for the heap [9, 85] and stack[20, 89] have already seen a significant amount of work in recent times and has been shown to be easily and effectively applicable. Alternatively, programs could also sprinkle arbitrary tokens across the data region in a configurable manner to catch such attempts. Furthermore, randomization could also be added to the order in which quarantined memory is released in order to augment temporal safety provided by this technique.

• **Temporal Protection.** In terms of temporal safety, ASan's, and consequently our guarantees are incomplete since we unmark previously allocated blocks when we reallocate them, after which point, dangling pointers or double frees can no longer be detected. This can be prevented to some extent by using heuristics such as reducing reallocation predictability by maintaining some degree of randomness for new allocations and ensuring that its entropy is never compromised by maintaining a large enough free memory pool. In our setup, however, we rely on ASan's existing allocation algorithm and do not augment it any further.

• **Composability and Coverage.** In order for ASan to be effective, *all* memory accesses to user data need to be monitored. Hence, it is essential that all software modules (the main program and shared libraries) be compiled with ASan support. Consider a situation where the program itself has been compiled as desired but a third-party library has not. In such a case, if the library has

Core	Frequency	2 GHz
	BPred	L-TAGE, 1+12 components, 31k entries total
	Fetch	8 wide, 64-entry IQ
	Issue	8 wide, 192-entry ROB
	Writeback	8 wide, 32-entry LQ, 32-entry SQ
Memory	L1-I	64kB, 8-way, 2 cycles, 64B blocks, LRU replacement, 4 20-entry MSHRs, no prefetch
	L1-D	64kB, 8-way, 2 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 4 20-entry MSHRs, no prefetch
	L2	2MB, 16-way, 20 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 20 12-entry MSHRs, no prefetch
	Memory	DDR3, 800 MHz, 8GB, 13.75ns CAS latency and row precharge, 35ns RAS latency

Table 4.2: Simulation base hardware configuration.

faulty code resulting in buffer overflow and it operates on a ASan-augmented buffer, the scope for exploitation still remains since read/writes in the library are not being monitored. The reverse situation also applies when the fortified code is in the ASan-augmented program but the data originates in the library, since the foreign buffer does not have the right bookends. Hence, ASan requires both access monitoring and metadata maintenance, one or both of which might break when using non-ASan augmented modules. Analysing and instrumenting the shared libraries at runtime would incur a huge performance penalty (as demonstrated by tools like Valgrind [83])[5].

REST relaxes this requirement greatly by not requiring explicit access monitoring. Thus, as long as the data itself is properly bookended, it does not matter whether the code accessing it has been instrumented or not. As such, it is more compatible with untreated external libraries. Since token access also generates exceptions at higher privileged levels, token manipulation via syscalls is also prevented.

4.6 Evaluation

4.6.1 Performance Overheads

We implement *REST* in the out-of-order CPU model of gem5 [13] for the x86 architecture. Due to its limited support for large memory mappings, we were unable to run x86/64 binaries since gem5 could not accommodate ASan's shadow memory requirements. Consequently, we simulate 32-bit i386 binaries of the SPEC CPU2006 C/C++ benchmark on the modified simulator in the syscall emulation mode with a configuration shown in Table 4.2. The `arm` and `disarm` instructions were implemented by appropriating the encodings for x86's `xsave` and `xrstor` instructions respectively, which are themselves unimplemented in gem5.

The benchmarks were compiled with Clang v5.0.0 with `"-mno-omit-leaf-frame-pointer -fno-optimize-sibling-calls -fno-omit-frame-pointer -mno-sse -O3"` flags. We run these programs to completion with the test input set. Since executions with these inputs spend a significant amount of time initializing (and allocating) compared to the ref input set, this choice of input sets should reflect on our results adversely since the overheads associated with our allocator will not be amortized with computation as well as in the case of ref inputs.

To evaluate *REST*, we compare it against two baselines — unsafe, plain binaries using the stock `libc` allocator, and binaries fortified with ASan. We evaluate two modes, secure with imprecise exception and debug with precise exceptions, for two defensive scopes, full (i.e., stack and heap) and heap only. Additionally, we present another category of numbers for perfect, zero overhead *REST* hardware (referred as PerfectHw) as a limit study of the current hardware design's optimality. The results are presented for each benchmark in Figure 4.8 as slowdowns relative to the unsafe binary.

***REST* vs. Baseline.** In the secure mode, *REST* shows an overhead of 26% and 22% while providing full or heap safety respectively. For the debug mode, the corresponding values are 71%

[5] ASan mitigates this to some extent by intercepting common library calls (like `strcpy`), checking the input data appropriately before the call.

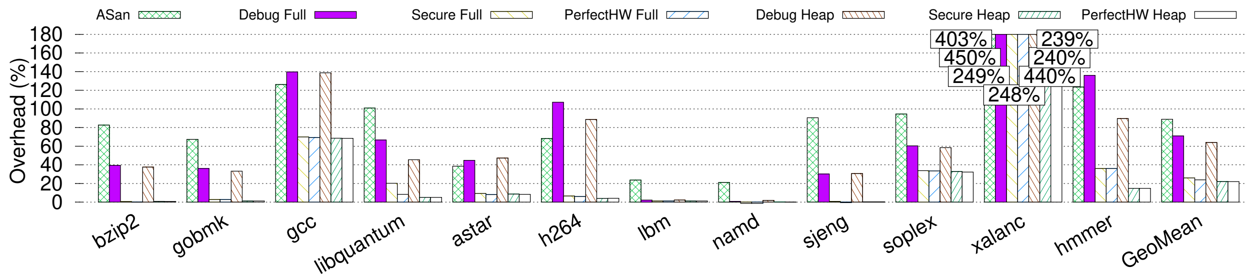


Figure 4.8: Runtime overheads (over plain) of ASan and *REST* in the debug, secure, and perfect (zero-cost) hardware modes while providing full and heap safety.

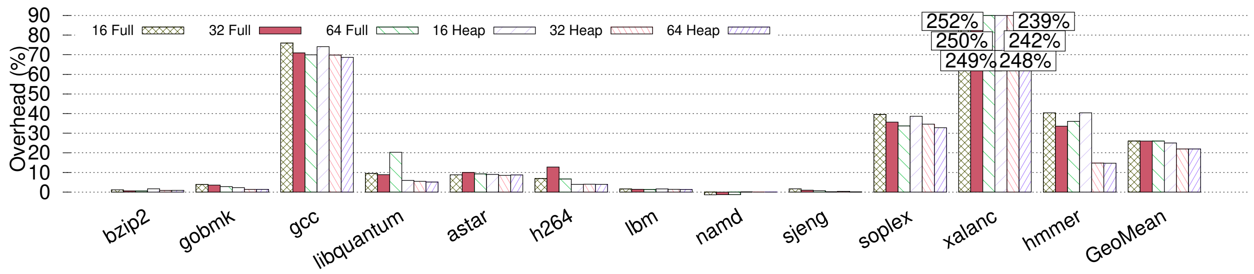


Figure 4.9: Runtime overheads (over plain) of using 16B, 32B and 64B tokens in secure mode.

and 64% respectively. In both modes, we find that the overall trend is roughly consistent with the results presented in Figure 4.3. Relative to ASan, *REST* does not perform memory checks (via explicit program instrumentation or `libc` call interception). In case of just heap safety, it additionally does not bear the cost of stack instrumentation. Accordingly, we observe that the numbers for *REST*'s full safety follow the expected trend. `gcc` and `xalanc` exhibit especially high overheads since they use the allocator more frequently than others (as also indicated in Figure 4.3), which provided the breakdown of various components of *REST*'s slowdown. Especially in the case of `xalanc` which makes a high frequency of allocations (0.2 allocations per kilo-instructions), the allocator overheads dominate significantly compared to other benchmarks. Benchmarks that use the allocator more sparingly (`lbm` and `sjeng`, for instance, which make less than 10 allocation calls overall) have little to negligible overheads.

These results additionally indicate that our allocator, based on ASan, is a major contributor to *REST*'s overhead. This is evidenced by the fact that the full and heap safe categories exhibit almost equal overheads, differing only by 0.16% on average. Thus, if recompilation is an option for users, *REST* could provide stack safety at nominal extra cost. We chose to use the ASan allocator for convenience; in the future, we plan to design a custom *REST* allocator that could potentially mitigate some of the observed overheads.

The difference in runtimes for the secure and debug modes arises due to the fact that, in the debug mode, we delay store commit until the corresponding write completes. In our simulated out-of-order core, although the impacts of this change manifests in many ways, a few side-effects were predominantly observed. First, unsurprisingly we found that the number of cycles the ROB was blocked by a store was about an order of magnitude higher in the debug mode. IQ occupancy was also severely affected for the latter case, especially for `xalanc` that had the number of cycles IQ was full in the secure and debug modes differed by more than 100x. Notably, we also did not observe a lot of traffic at the main memory interface due to token fills, indicating that most token accesses hit in the cache and do not otherwise contribute to memory access bandwidth for any of the benchmarks in either mode (only 0.04 tokens per kilo-instructions crossed the L2/memory interface for `xalanc` in the secure full run).

Software vs. Hardware. To distinguish between the overheads added by our software and hardware modifications, we run the *REST* binaries on stock hardware with one key modification — each arm and disarm in the binaries is replaced by one regular store. This simulates a situation wherein our *REST* hardware modifications for managing and checking tokens have zero cost. The runtimes for this set of experiments are shown in Figure 4.8, denoted by the *PerfectHW Full* and *PerfectHW Heap* bars. As these results show, the overheads incurred by the perfect *REST* hardware are not significantly different from that seen in the secure mode, being only 0.2% lower for full protection and for heap protection. This implies that the cost of the *REST* primitive in hardware is nearly zero and that the entirety of the performance overheads in the secure mode are solely an artifact of its software component, especially the allocator.

Token Widths. Token widths while affecting the security of a system might also potentially affect its performance, since smaller token widths might allow better cache utilization. In order to evaluate this we configure our implementation to utilize tokens of 16B and 32B and perform the

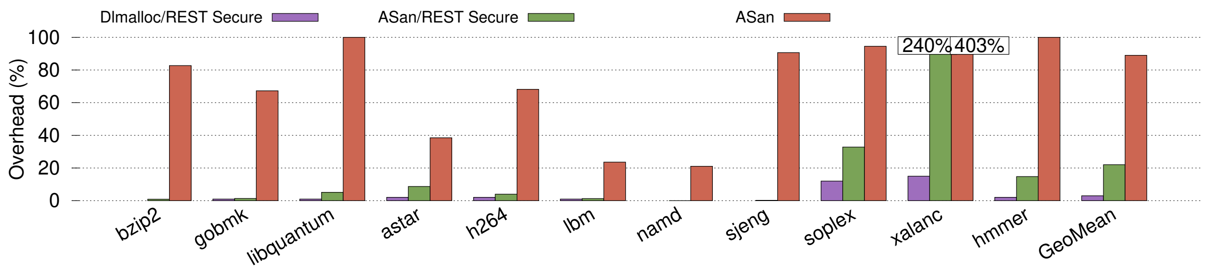


Figure 4.10: Runtime overheads (over plain) of heap protection with dlmalloc. ASan/REST and ASan results are also shown again for reference.

experiment for all modes. The corresponding results are shown in Figure 4.9. Overall, we see that choosing any single token width does not make a significant difference in terms of performance. In the general case, users might thus freely choose robustness in the form of wider tokens, without compromising performance.

dlmalloc vs. ASan. We also ran the above experiments with our modified `dlmalloc` for heap safety. Our baseline for these experiments was the plain binary run with regular `glibc` with the original `dlmalloc` as the allocator. These results are shown in Figure 4.10. We notice that the overheads are significantly lower for all benchmarks. Of note is `xalanc`, whose overheads drop dramatically from 249% to 15%. With the new allocator, our overall slowdown drops to less than 4%.

4.6.2 FPGA Area Overheads

We implemented the *REST* modifications on the Leon3 SoC package, which consists of a SPARC32 V8 processor. Leon3 has one level of split data and instruction caches of 16B width, which are not write allocate. So, for our modifications, we first change Leon3's data cache to a write allocate version, and implement the *REST* changes on top of it. For the sake of this analysis, we consider the Leon3 cache with our write-allocate modifications to be our base design. The design was then synthesized and implemented on a Xilinx Virtex-7 based VC707 board.

Overall, we added less than 200 lines of VHDL to the data cache's RTL. The modifications did not affect timing and added ~100 LUTs to the synthesized design.

4.7 Related Work

Memory safety implies two types of protections — spatial and temporal. Spatial memory errors usually manifest in two different ways depending on program behavior. Overflow-style errors are a result of a sweeping or *linear* access pattern wherein the code sequentially starts accessing locations beyond the bounds of the data structure. Alternatively, invalid reads/writes might also occur if a pointer is corrupted/overwritten resulting in a access pattern that can be more precise or *targeted*. Protection schemes can be characterized depending on which pattern they detect. In terms of temporal protection, schemes can be characterized by the time window within which their protection lasts. Some schemes provide *complete* protection by detecting dangling pointers for the duration of the entire execution, while others only do so until the invalid region has been reallocated again.

Since memory safety has been a persistent problem for decades, a lot of work has been done to address it, especially in software [114]. In this section, we only discuss relevant hardware techniques proposed towards solving this problem (summarized in Table 4.3) below.

• **Bounds Checking.** Hardware-based bounds checking [36, 45, 79, 80] solutions were proposed to mitigate the problems of high performance overhead associated with software enforced bounds-checking [54, 81, 82] while retaining its effectiveness. They were quite successful in this regard, bringing down the performance penalty significantly (Hardbound [36] reported considerably lower overheads than the others but does not provide temporal safety). There are a few differences between them and *REST*, however. This is because of the fact that while bounds-checking performs complete monitoring of out-of-bounds accesses (assuming pointer identification in hardware is perfect), *REST* only detects errors when the blacklisted locations are accessed and hence, provides weaker security guarantees. The advantages of the latter approach, however, are lower overheads and complexity.

Firstly, *REST*'s memory overhead scales with the number of protected data structures, not pointers to them, and does not need separate memory to do so. We also do not require storage in the chip itself, other than a register at the L1 data cache. On the other hand, most previous works store metadata in a shadow space, a memory region containing metadata for every location of the address space. This results in fast metadata access since calculating its location inside the shadow space is derivable by a simple arithmetic operation on the pointer address. But it is also highly inefficient in terms of storage since all of the address space is shadowed even if a negligible fraction of it is actually occupied by pointers. Watchdog [79] and Watchdoglite [80] reported ~56% increase in memory usage for SPEC CPU benchmarks. In terms of on-chip storage, all schemes, with the exception of Watchdoglite, introduce some form of fast-lookup memory, such as caches, in order to speed up metadata lookup and hence, pointer operations.

Most of these schemes also introduce non-trivial hardware logic to the chip microarchitecture. Hardbound and Watchdog inject micro-op around memory accesses instructions at runtime. SafeProc and Watchdoglite, on the other hand, rely on the compiler to explicitly insert instructions in the program to this end, enabling static analyses to optimize these operations. Furthermore, Watchdog logically extends the physical register file to accommodate metadata, whereas the others use existing registers, thus increasing register pressure. *REST*'s detection logic is vastly simpler since we do not perform checks for spatial and temporal violations in the pipeline for every memory access. Since we defer the detection responsibilities completely to the caches, the core architecture itself remains unchanged, also making register pressure a non-issue.

Additionally, reliance on compiler support implies these systems have limited composability with software (such as third-party libraries) which have not undergone the necessary static transformations. This means they necessarily require shared libraries that have been compiled similarly. Critically however, a kernel that is unaware of this scheme could cause errors and presents a potential vulnerability for such systems. For instance, an attacker could influence the size arguments of a data-manipulating syscall to corrupt sensitive data. Since *REST* associates metadata with the data structure and not its pointers, we do not have to worry about static pointer analyses (or their accuracy). The compiler support necessary for *REST* is, hence, significantly simpler (LLVM's ASan module has only 2129 LoC with our modifications).

Notably, Intel Memory Protection Extensions (MPX) [53] marks the first commercial support for this technique. However, it faces a few compatibility issues and exhibits high performance overheads [86].

• **Tagging.** Some defenses "color" memory regions by associating tags with them and checking these tags when they are accessed. HDFI [107] marks memory locations with a 1-bit tag, that subsequently indicates whether that location can be accessed via regular load/stores. Although it is quite flexible and exhibits nominal overhead, its hardware requirements are higher than ours. SPARC ADI [48] uses a 4-bit coloring scheme, using the 4 most significant bits of a 64b pointer for this purpose. On an access, the hardware checks whether the tags of the pointer and accessed regions match. They also require a custom allocator responsible for coloring heap allocations but do not require that programs be recompiled to avail this feature. Although full details of the microarchitecture have not been disclosed, at a minimum they require 4 bits of metadata per cache line at all cache levels. Spatial overflows are prevented by annotating adjacent allocations and their metadata with different tags, while temporal overflows are prevented by changing tags on deallocation. However, due to the limited number of available tags, memory regions might reuse tags after being reallocated enough times (via heap feng-shui attacks [108], for instance) after which dangling pointer access will go undetected. Moreover, since they modify pointer format, (legacy) programs that do special pointer operations involving compression or irregular arithmetic will be incompatible with this technology. We do not face these problems.

• **Capabilities.** Capability-based architectures [120, 126] are another metadata-based secure hardware design that offer stronger security guarantees than us. Here, all pointers are augmented with metadata that goes beyond bounds information (permission, for instance). Particularly, works in the CHERI project [22, 126] have demonstrated its applicability in the modern era on a whole-system level, not just for applications, for a MIPS 64-bit in-order processor. However, this support comes at the expense of high performance and area overheads, although the authors acknowledge open areas of optimization in their design.

• **Watchpoints.** This class of solutions aim to provide a high number of hardware data watchpoints, primarily for debugging. iWatcher [128] was one of the first hardware techniques proposed to this end and functionally provided support for a high (but limited) number of programmable hardware watchpoints at a relatively low overhead compared to some software solutions, but required that the affected physical pages be pinned to physical memory and not be swapped out. Although they did not explore memory safety as an application, Greathouse et al. [47] solved both problems by providing unlimited watchpoints and allowed pages to be swapped out by storing metadata separately.

• **Others.** SafeMem [96] repurposed error checking ECC bits in main memory to mark memory locations invalid in order to detect spatial memory errors. They did so by setting the parity state to an error value, so that accesses to those locations trigger exceptions, thus trading reliability for safety. However, each set/unset operation is quite expensive with latencies comparable to an `mprotect` syscall. Additionally, it did not support the swapping main memory contents to disk. Memtracker [119] associates state with each memory location by monitoring accesses to them. They however, do not make any modifications to the allocator to inhibit allocation reuse, and so are more vulnerable to temporal attacks. Besides the above solutions, ARM recently announced pointer authentication in select chips [97] that counter pointer corruption and forging, but do not protect against general temporal or spatial attacks.

4.8 Conclusion

In this chapter, we proposed *REST*, a primitive for content based checks and showed how it can be used to creae a low complexity, low overhead implementation for improving memory safety. *REST* itself requires local modifications that integrates within existing hardware intefaces. It incurs a low performance penalty for stack and heap safety, which is 22-90% faster than comparable state-of-the-art software implementations, while additionally being more secure and providing heap safety for legacy binaries.

There are many open areas of optimization and extension to *REST*. The *REST* software components viz., the repurposed Address Sanitizer allocator, accounts for almost all of the slowdown in the secure mode. An allocator designed to take advantage of *REST* properties and requirements could be significantly faster. Similarly, for hardware, our goals was to minimize number of optimizations: however, a few additional microarchitectural optimizations such as a dedicated cache for *REST* lines has potential to decrease overheads further, especially for the debug mode and for programs that make frequent allocations. Finally, we only explore *REST* at the application level in this chapter; extending and supporting it at the system level and for heterogeneous architectures, will increase system security and reliability.

The benefits of *REST* go well beyond memory safety. As a primitive for performing content-based checks in hardware, it provides a number of opportunities not only for improving other aspects of software security (e.g., control flow), but also programmability and performance. Developing these new applications using *REST* can bring significant exciting benefits.

Proposal	Spatial Prot.	Temporal Prot.	Shadow Space	Composability	Perf. Overhead	Hardware Modifications
Hardbound [36]	Complete	None	✓	✗	Low	μop injection, L1 cache & TLB for tags
SafeProc [45]	Complete	Complete	✗	✗	Low	Multiple CAMs and memory units, hardware hash table, hash table walker
Watchdog [79]	Complete	Complete	✓	✗	Moderate	μop injection, pointer lock-ID cache, dangling pointer monitor
Watchdoglite [80]	Complete	Complete	✓	✗	Moderate	Nominal
Intel MPX [53]	Complete	None	✗	✗†	High	Not known
HDFI [107]	Linear	None	✓	✓	Negligible	Wider buses and cache lines, tag-aware memory controller with caches, tag table
ADI [48]	Linear‡	Until realloc‡	✗	✓	Negligible	4b per cache line at all cache levels‡
CHERI [126]	Complete	Complete	✗	✗	Moderate	Capability coprocessor tightly integrated with in-order pipeline
iWatcher [128]	N/A	N/A	✗	✓	High	Per-byte cache line metadata, a multi-entry table, small metadata victim cache at L2
Unlimited watchpoints [47]	N/A	N/A	✗	✓	High	Range cache, metadata TLB
Safemem [96]	Linear	None	✗	✓	High	Repurpose DRAM's error-correction bits
Memtracker [119]	Linear	Until realloc	✓	✓	Low	Metadata caches, monitoring unit in pipeline
ARM Pointer Authentication [97]	Targeted	None	✗	✓	Negligible	Not known
REST	Linear	Until realloc	✗	✓	Moderate	1 metadata bit per L1-D line, 1 comparator

Table 4.3: Comparison of previous hardware techniques (assuming single-core systems for simplicity). †Although MPX-supported binaries execute with modules that are not protected, metadata is dropped when such modules manipulate an MPX-augmented pointer. ‡See text.

CHAPTER 5

Address Space as a Primary Line of Defense

The virtual address space is a fundamental abstraction in computer systems, the properties it grants and its implementation being uniform across architectures. Programs executing in this environment can use virtual addresses to access content in any location, as long as these properties are not violated. Valid generation and usage of addresses follow strict but simple rules, such as linear progression of addresses and page granularity of permissions, that are derived from the same properties. Software attacks, which run within the context of the program, are also bound to these same rules and abuse them at runtime to achieve their goals.

In this chapter, we argue that these rules of generating valid addresses are overly liberal. Instead, we propose and define an alternative address space abstraction, called the apparent address space (AAS), that makes statically unintended usage and generation of addresses harder at runtime, thus limiting the capabilities of runtime attacks. Subsequently, we propose some hardware implementations for this idea, and conclude that, although the idea may sound novel in theory, the implementation cost does not justify the benefits, when viewed as a standalone defense mechanism.

5.1 Introduction

Current principles of program execution are tightly coupled to the notion of an address space. In this respect, "address" is a form of information used by the program. While most addresses are statically baked into the program, it is also made available dynamically (to a degree that is architecture-dependent). Like the program itself, most attacks leverage this very fact by, implicitly or explicity, extracting and using address-related information available at runtime via what we term the *address-interface* to bypass defense measures.

Virtual address space (VAS) is a highly entrenched concept in modern system and application design, wherein each program operates in an isolated, sanitized view of its accessible memory. This view of the VAS has an implicit property — a system-wide consensus that address progression is a linear sequence. Thus, the "next" location can be calculated by simply incrementing the current address, whereas the "previous" location is the result of a simple decrement. Formally, given a location x, the locations before and after can be calculated with the following *sequence-functions*.

$$\begin{aligned} previous(x) &= x - 1 \\ next(x) &= x + 1 \end{aligned} \tag{5.1}$$

Thus, code executes in a straight line (i.e., fetches instructions from subsequent memory addresses) unless explicit control-flow change occurs, and data of size greater than native types (strings, structs, etc.) are arranged contiguously. This agreement allows various architectural, system, and compiler optimizations to be engineered towards efficient and performant program execution.

The fact that virtual-addresses progress in a linear sequence can, however, be exploited allowing an attacker to blindly probe or stride the address-space in order to leak or corrupt program state. Consider the classic attack of return address corruption via a buffer overflow vulnerability. In this case, the attack overflows the exploitable buffer to reach the return address slot. While this is a result of the operational design of stack frame layouts, it can also be generally ascribed to the fact that programs are architecturally presented a flat view of the virtual address space. Underlying these attacks is the implicit knowledge of the fact that not only are the sequence functions publicly known, but they are also quite simple and uniform, even across architectures. This gives rise to unintended and traversable proximities among locations of memory objects (code or data), that are not always a program requirement and exist solely as an artifact of the linear nature of the address space. The only hard deterrent to invalid linear accesses is enforced at the page granularity, which in many cases is too coarse and easily surmountable.

To make matters worse, addresses are also often embedded into the program itself or injected explicitly or implicitly. When leaked, these can then provide the basis for attacks to access other adjacent locations using the VAS sequence functions.

In this chapter, we posit that granting this degree of latitude when it comes to specifying and enumerating addresses is not only overly simplistic and unnecessary, but even harmful from a security perspective. To mitigate these problems while keeping implementation costs low, we propose the apparent address space (AAS) as an abstraction layer over the traditional virtual address space (VAS). As with the physical and virtual address abstractions, programs can only view the AAS, while the system backend is aware of the VAS. Additionally, the AAS possesses certain properties that makes implicitly imposes memory protection at a fine granularity. As we attempt to deal with the challenges of a hardware-software design of this, we, however, realize that the benefits it provides relative to other works in the area are not commensurate to the complexity of its implementation.

5.2 Motivation

In this section, we define the address interface in more detail, outlining how it is a seldom considered but crucial aspect of program execution as well as software attacks. We will then propose an address space abstraction that significantly reduces the scope of this interface, making it hard for statically unintended behavior, as is leveraged in software attacks.

5.2.1 The Address Interface

Address is an architectural alias for identifying locations and their contents uniquely. In the context of user-space programs, this corresponds to the virtual address. We define the runtime address generation interface (AGI) as the means by which a program can derive and/or generate addresses at runtime. There are broadly two aspects to it.

Injected Addresses. First among these are the addresses embedded or injected into the program directly. These are the primary sources of addresses in the program and can be introduced into the program by the following means.

- **Via compilation and loading.** The compiler framework embeds jump offsets and address variables in the program. Part of this responsibility is also shared by the loader for position-independent code and dynamic loading.

- **Via syscalls.** Syscalls are the conduit for processes to interact with the outside world and manage available resources. As such, the OS will often introduce new, valid addresses into a program as defined by the syscall interface. For instance, heaps are initially allocated through a syscall (`mmap` or `brk` on Unix based systems) and subsequently managed by a memory manager.

- **Via architectural vectors.** Some architectural registers, like the frame-pointer, stack-pointer, and program counter, may also expose addresses to the program, directly or indirectly. Additionally, some ISA operations, like function calls and exceptions, also inject valid addresses into the process' purview.

Generated Addresses. A secondary source of addresses in the program are the new ones that are generated from primary addresses noted above via the sequence functions. This is a critical property of the AGI because it allows the program to represent and specify contents of non-native sizes with a single "name". Hence, an array can be represented with the address of the first location; the location of any subsequent element is obtained as an offset from the head element.

Similarly, an entire function of code can be represented with just the function address. Given this one representational alias, the locations of the rest of the content (e.g., other elements in the array or instructions in the function) can be calculated using the sequence functions. Without them, non-native memory objects would have to be individually specified, making programming as we know it infeasible. The VAS, regardless of underlying architecture, universally follows the simple sequence functions defined in Eqn. 5.1.

Thus, the AGI can be completely specified using the explicit address sources and sequence functions mentioned above, with minor variations depending on the platform, ABI, etc.[1]

5.2.2 Exploiting the AGI

Since software attacks work within the context of the program, they also use the AGI. Depending on the vulnerabilites and vectors, they do so using one or a combination of the following techniques:

① **Leak valid address.** Memory disclosures can occur as a result of a myriad of reasons, ranging from programming bugs to language stipulations to architectural ABI regulations, or a combination of them. Although many defenses have been proposed against them, the sheer variety of their causes makes coming up with comprehensive generic solutions a hard endeavor.

② **Hijack one or more of the program's address usage gadgets.** Often vulnerabilities result in attackers being able to manipulate memory access code snippets in the program. For instance, an attack could utilize an overflow bug in a program loop to make out-of-bounds reads/writes. Alternatively, if he can control the input address to the loop, program state can be easily leaked or corrupted [50].

③ **Inject/reuse code.** With this powerful capability, the attacker is able to inject arbitrary code into the program or execute snippets of the program's own code to achieve the same functionality. Getting to this point is generally preceded by one or both of the previous steps.

Even with memory disclosures, malicious attempts at reading or modifying the targeted, sensi-

[1]Since we consider contents at the machine, raw memory level, we discount languages from the discussion.

tive memory objects rarely pan out in the very first step and in practice, proceed in stages employing any or all of the means stated above. Regardless of the methods or operational requirements, attacks usually try to "reach" the object of interest from another object that the attacker did gain access to or can control. This general principle of traversing is widely applicable and used both in the wild as well as in research. For instance, this theme is the fundamental premise of all overflow attacks. The Heartbleed bug [31], affected OpenSSL 1.0.1, used a classic buffer overflow vulnerability using the innocuous heartbeat object to reach more sensitive data. Alternatively, a widely used category of egg-hunting shellcode [77] scans the address space for the injected code to jump into. In the research domain, some recently published attacks [14, 106] scan the code memory to dynamically harvest gadgets in order to execute a code-reuse attack [101].

Since memory disclosures themselves are a shifting gap to plug due to the widely varying nature of its causal factors, the AGI is a more promising target for hardening, especially since its properties are highly uniform and widely applicable. Hence, in line with the principle of least privilege and information, the ideal goal of securing this interface is to reveal only those addresses necessary for execution, as foreseen by the developer and encoded in the program semantics. Any other address value should not be exposed via injection or in-program generation.

5.3 Related Work

Randomization is popularly considered a measure to counter predictable program layout. However, most of recent proposals in this direction have targeted code at different granularities [63]. Some schemes have specifically targeted the heap [9, 85] and the stack [20, 89], while a few schemes have targeted data layout in general [10, 18, 67]. Even so, the coarse granularity of page permissions and their static nature implies that static randomization can be subverted with runtime memory scanning [106]. The recently proposed hardware based data randomization scheme, HARD [8], creates equivalence classes of data with context sensitive static analyses and encrypts the different classes separately. It is, hence, somewhat resilient to runtime scanning, but still suffers from pathological problems arising from inaccuracy of the compiler analysis, in which case pointers to equivalent data structures can point to any structure in the class.

Bounds-checking is also an oft-explored idea in this regard. Software versions of it are ineffective against the adversary outlined by our threat model. Hardware enforced fat-pointers [36, 45, 79, 80] are not only expensive and complicated, but are based on a flawed assumption – the attacker always receives a properly created pointer and cannot otherwise create or manipulate the creation of one. This is not the case for our powerful adversary. Capabilities [120, 126] have also been shown to be effective in this regard. But they require significant hardware re-engineering, so we will discount them from this discussion.

Alternatively, hardware tripwire schemes [96, 105, 107] implement the much simpler primitive allowing the program to mark certain memory regions invalid. The program then utilizes it to mark certain data or the memory around it invalid. Although these techniques present very low overheads, they can be easily bypassed by simply jumping over the invalidated regions (assuming their locations are known) while traversing the address space.

5.4 Apparent Address Space

Seeing how the AGI can be abused by attackers, it becomes pertinent for us to judiciously expose it to the program. In order to limit the degree of information available through it, we propose the *apparent address space* (AAS), which is an abstraction over the virtual address space but does not expose sequential addressing (as shown in Eqn. 5.1). Instead the AAS exhibits the following sequence functions,

$$\begin{aligned} x_{\text{AA}} &= f_{\text{key}}(x_{\text{VA}}) \\ previous(x_{\text{AA}}) &= f_{\text{key}}(x_{\text{VA}} + m) \\ next(x_{\text{AA}}) &= f_{\text{key}}(x_{\text{VA}} - m) \end{aligned} \tag{5.2}$$

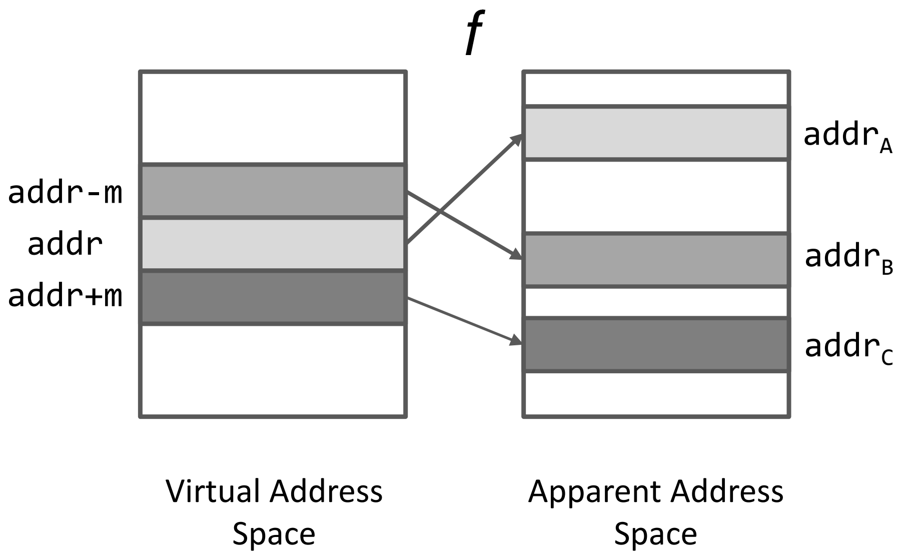


Figure 5.1: Apparent Address Space as an Abstraction over the Virtual Address Space.

, where m is the sequence granularity and f is the hashing function that uses the key, `key`. The algorithm of f is publicly known, but `key` is secret at the user level. It is, however, known to the compilation framework and runtime back-end including the hardware.

Imposing such a stipulation over the program conservatively limits the capabilities of any entity operating within it, malicious or benign. Furthermore, keeping f a secret also has the following side-effects.

① *Given the location of one or more memory objects, the location of another is not derivable by statistical or empirical correlation (inference property).* f essentially "randomizes" the address space at the granularity of m. Hence, this scheme subsumes the benefits of fine-grained randomization for both code and data. Assuming f is irreversible, this property prevents attackers from directly inferring addresses of memory objects adjacent or semantically unrelated to the ones directly under control. Note that this does not affect a (well-structured) program however, since legitimate code explicitly encodes the data flow so that it has address of the memory object when necessary.

② *Attempts at accessing a location blindly are forbidden and/or caught (whitelisting property).* The VAS (at least for 64-bit addres spaces) is quite sparse, so that most locations actually map to unallocated pages. Assuming f uniformly hashes all addresses, most AAs should, hence, correspond to invalid addresses. Furthermore, if m is small enough this property essentially prohibits blind address-enumeration. This is significant since given that the attacker lacks the capability of deriving locations of objects without leaking them directly, this property prevents the alternative, which is to employ some guided probing mechanism to reach other possible locations. Thus probabilistic attacks techniques like memory scanning and buffer overflows are ineffective. Again and for the same reason as before, the actual program should be unaffected by this property.

5.4.1 Security Implications

As a corollary, the two properties outlined above in turn enforce the following property on the program.

Exact address of a memory object must be known to access it.

Incidentally, this also outlines the goal of spatial memory safety, which essentially treats every pointer as a unique key which can open the lock to and access only the data structure it points to. In other words, a pointer to a memory object may only dereference to the memory region within object. The AAS functionally achieves the same because of the *whitelisting* property. It additionally goes beyond memory safety with the *inference* property, wherein an ideal AAS also makes it impossible to create new pointers to valid memory unless the exact memory location is known. Traditionaly memory safety definitions do not regulate pointer creation (although capability based architectures do).

While the effect of the above property on data is clear, applying them to code has an interesting consequence. Specifically, if a code address is disclosed (by leaking a function pointer, for instance), addresses of subsequent (or previous) instructions cannot be enumerated, thus obviating code scanning and attacks that rely on it [106].

5.4.2 Challenges

There are two major lines of challenges that a practical implementation of AAS would have to face.

• **Hardware Based Challenges.** The hardware-software system hierarchy has seen decades worth of optimization aimed at streamlining VAS functionality. As such, a system design from scratch to support the AAS is untenable. Hence, the first challenge is to maintain the current architecture as much as possible and implement the AAS on top. Our decision to implement AAS as another abstraction level on top of the VAS would largely take care of this concern. All this would entail, then, is to perform the AA-VA translation at the appropriate interface, thus mostly obviating design changes in the software and hardware infrastructure behind. We discuss particulars in the next section.

• **Programming Construct Based Challenges.** A critical aspect of programs that we have avoided discussing so far are sequential data structures (henceforth referred to as arrays, for simplicity). The main challenge of AAS is the following: how do we represent arrays, which are semantically sequential, with a single representational alias (or name) in the fundamentally non-sequential AAS? Arrays are an unavoidable construct of programming languages and a fast representation for them is critical. As the traditional addition-based offsets no longer apply in the AAS, an alternative is essential for practical feasibility. Simultaneously, we have to make sure that this alternative cannot be abused by an attacker to access illegitimate locations in the program. In other words, we have to ensure that any concessions or mechanism granted by the AAS does not give attackers a means to bypassing its protections and thus, violating its core properties.

One obvious way of enforcing the *inference* property is by employing layout randomization comprehensively on every aspect of the program (code and data), at as low a granularity as possible. Owing to the multitude of randomization techniques available, this would require usage of multiple schemes. Even so, due to the flat nature of the address space and the fact that memory objects are packed together, this solution would be vulnerable to memory scanning, allowing cross-over from one object to another. The key reason for this is that there is no detection mechanism built into such a system.

Techniques that satisfy the *whitelisting* property could be the solution to this problem. Inserting large redzones or invalid pages between memory objects could solve this, but this has a high memory overhead. But they can just be jumped over if an attacker has prior knowledge of their presence. Additionally, some syscalls allow the program to probe whether an address lies on an invalid page without crashing the program (see [77]). Alternatively, a bounds-checking scheme could also prohibit cross-overs. However, such schemes do not seek to uphold the *inference* property by hiding pointer values. Furthermore, the hardware does not validate the bounds aggresively by authenticating whether they actually correspond to the object being pointed to, thus compounding the problem. So, once an address is leaked, the attacker could infer the location of other objects and subsequently control/create pointers to access them. In other words, pointer forging and manipulation are theoretically possible, which would violate the *whitelisting* property. Hence, we see that while there already exists ways of partially achieving the AAS properties, none satisfy both entirely, leaving fundamental loopholes in the design.

5.5 Implementation

In this section, we will outline a few design options for implementing AAS with different tradeoffs. Notably, our designs will be based on already existing technologies, so integration costs can be minimized.

5.5.1 Hardware Design

The first design point addresses how we enforce the sequence functions (Eqn. 5.2) and the hashing function, f. Since address translation is a very common phenomenon during execution, ideally f should be able to complete within a single cycle so that the timing requirements of some pipeline stages are not stressed. For our purpose, we choose f to be a combination of the XOR and/or bit scrambling. Notably, ARM's pointer authentication already allows pointer encryption with some low-overhead encryption schemes [97], which we could utilize[2]. Every binary would have a key associated with it, that will be used to encrypt/decrypt addresses as specified by Eqn. 5.2. For simplicity, the encryption granularity, m, can be equal to the native data width.

The design thus far establishes AAS as an abstraction over the VAS, so that the actual memory layout of the program, the supporting software and hardware infrastructure do not have to change much beyond the address decryption around the execute stage of pipeline. As we discuss later, some additional architectural modifications also have to be made regarding pointer operations. Hence, in order to support arrays, we change the way addition based offset calculation to include decryption and encryption before and after.

However, this still does not solve the problem of overflows. Any mechanism used to calculate offsets within an array can also be used to calculate addresses outside, enabling overflow situations. This, therefore, essentially allows the program a glimpse of the VAS and the layout within, thus violating the *whitelisting* property at the least, and also the *inference* property, depending on the context.

To solve this issue, we utilize a tagging mechanism, similar to ones recently announced by SPARC [48] and ARM [3]. Consequently, we require that arrays or any objects bigger than the native data size be tagged with a "color", and all pointers to those objects carry the color in them. A pointer dereference is only valid if it is to a memory region of the same color (as shown in Figure 5.2). Colorless pointers are used for native data types and should not be operated on beyond the native width. As far as hardware overhead is concerned, the above techniques only support 64-bit architectures and embed the pointer tag into the higher order address bits of the pointer value. Furthermore, memory is also logically extended to associate tags with each location, with which the pointer tags are matched. So far, these are common features among comparable tagging mechanisms.

One corner case that presents a slight complication in our design is that of pointer comparison.

[2]However, their encryption requires an additional *context* key, which we may not require. Instead, we could use a similar equivalent such as Prince [15].

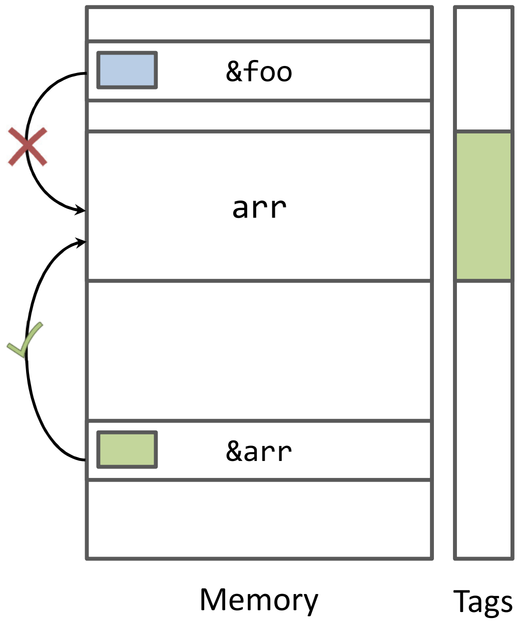


Figure 5.2: Tagging scheme employed for coloring memory objects in AAS.

In a linear VAS, this operation made logical sense, while in the AAS, it does not. Semantically speaking, pointer comparison only makes sense if its operands belong to the same higher memory object. In keeping with this line of reasoning, we allow pointer comparisons only if the two pointers have the same color.

Limitations. The VAS fundamentally does not support the entire C memory model due to its non-linearity. This includes addition-based pointer arithmetic and pointer comparisons as noted above. By the C pointer model, pointers are also allowed to be modified as long as they point back to the right area at dereference. Our design does not accommodate this for native pointers. Additionally, this design does not support dynamic linking in its current form.

Architectural Interface. The above modifications mandate a few changes in the architectural interface exposed to the program. Firstly, as with the base tagging architectures, tag creation and manipulation instructions have to be introduced. Secondly, new pointer arithmetic instructions also have to be supported with the properties stated above since traditional integer add, subtract, and logical compare operations would no longer be valid. Thirdly, all architectural addresses injected into program (through the stack pointer register, return addresses, etc.) have to be encrypted with the program's address-encryption key.

5.5.2 Software Support

To support the AAS, we need the following changes in the supporting software infrastructure.

Compiler and Linker. The compiler has to convert all variable accesses to pointer accesses and assign colors to them according to type, and change pointer arithmetic to use the special instructions described above. The stack epilogues and prologues also have to be modified to reflect this. Furthermore, all direct addresses, if any, have to be encrypted by the linker with the AAS key.

Operating System. The OS has to maintain the key as part of the process information and convert all incoming and outgoing addresses (via syscalls and interrupts) accordingly.

Note that the modifications outlined above implement a naive support. We did not seek to make any optimizations in this flow. For instance, it is possible to identify variables that are safe from being corrupted, and allow direct accesses to them instead of through a pointer to it. Exploring similar optimizations could yield significant performance improvements to the design.

5.5.3 Security Evaluation

• **Tag Width.** In an ideal implementation, there would be an infinite number of colors that can be associated with data structures, so that reuse of colors is never an issue. Practically, however, tag width limits how many colors can be assigned leading to color reuse. Color reuse is a risk since they can be exploited to mount temporal attacks.

• **Reversing f.** Depending on the robustness of the encryption function, f, it is possible to reverse it, especially if weak encryption schemes like XOR and bit scrambling are used. This could be easily achieved if one or more pointers to an array object can be obtained and enough addresses within it can be harvested. Hence, the exact scheme to use for f becomes a performance vs. security trade-off for the architect.

• **Pointer Hopping.** To have a pointer to an array object is to have complete access over it. So, a situation could arise where one obtains a pointer to an object, containing other pointers, which are then successively followed to reveal enough information about the program state. Our design of the AAS is ineffective against such an attack.

5.5.4 Feasibility

To understand the value of this technique, we have to ask ourselves what does this abstraction give us over current memory safety techniques? Specifically, for the sake of this discussion, if we assume we are leveraging the mechanisms provided by a tagging scheme like SPARC ADI, we have to determine how much of an additional security is added by AAS compared to a system that has ADI? From a practicality standpoint, this is important because performance-wise our scheme would be slower.

Memory safety in a program is a very strong defensive guarantee. Although the tagging schemes do not provide ideal memory safety, they are practically very secure, nonetheless. Added to this the fact that they have negligible slowdown, we unfortunately could not determine the practical security benefits for an AAS. This is despite the fact that purely in terms of hardware overhead and complexity, we would be comparable to present tagging architectures.

5.6 Conclusion

Addresses are a fundamental tenet of program execution, its usage guided by certain rules imposed by the virtual address space. Just like the processes themselves, software attacks follow the same rules at runtime to enumerate addresses according to the address generation interface. As such, securing this interface should seemingly go a long way in system security.

In this chapter, we argued for tightening of the guidelines governing address generation within the virtual address space in order to secure the program. To this end, we proposed an alternate view of the address space, called the apparent address space, whose properties automatically impose spatial memory safety on the program. Furthermore, we highlighted the challenges in achieving it and put forward an exemplary implementation it which utilizes established prior work and more importantly, does not require scrapping decades worth of optimizations aimed towards virtual addressing. However, the performance and implementation overhead introduced due to this abstraction was not deemed justifiable, especially compared to state-of-the-art in commercial memory safety hardware measures. It is possible, however, that novel attacks may be developed in the future that might necessitate such an approach. This is therefore an idea for the posterity.

CHAPTER 6

Concluding Remarks

The state of system security has reached an inflexion point wherein the traditional methods of defenses are unable to keep up with the conflicting usage requirements of providing protection against an ever expanding arsenal of threats, while managing to keep a low profile themselves. This is a great opportunity for hardware engineers to contribute in this domain, to which they have largely remained oblivious and insulated in the past. Both in research and industry, there is a realization that addition of appropriate hardware features could potentially make a significant difference from a security perspective. As welcome as this realization is, practical concerns dictate which techniques can be ultimately translated to commercial deployment.

To that end, in this dissertation, I assert that hardware support for system security, that have been traditionally implemented in software, need not be complex in order to be effective. In fact, I even argue that simplicity should be one of the primary design goals while engineering any defensive measure in hardware. In this dissertation, I have outlined three case studies to support this position. In the first case study, we secure code by designing simple hardware support for a classic technique called instruction set randomization to prevent state-of-the-art code reuse attacks, against which it was traditionally considered impotent. In the second case study, we propose a low-overhead, low-complexity hardware primitive, and use it to secure data against common types of memory safety violations. And finally, in the third study, we discuss a proposal to modify the address space to present a non-linear sequence of addresses to the program, but determine that the implementation overhead for such a design would unfavorably outweigh the cost of its security benefits, and thus violate the thesis outlined above.

6.1 Lessons Learned

Besides the points of simplicity and cost effectiveness made throughout this dissertation, I conclude with a few lessons realized during my studies that hardware engineers should consider and software engineers should be cognizant of while proposing and designing hardware defenses in the future.

Flexibility Aids Longevity. As emphasized repeatedly in the dissertation, the domain of security is constantly in flux, with ever newer threats and novel vectors for exploiting classic ones emerging at a break-neck pace compared to the rate at which hardware features evolve. Thus it is a considerable challenge for hardware security engineers to predict the state of security years into the future while proposing a defense that will be relevant for that period. In the worse case, a technique could be rendered useless either due to the availability of a better or comparable alternative software defense or just by the fact that the state-of-the-art for that threat has moved beyond the feature the defensive measure targets. One of the best ways to mitigate this problem and making the technique somewhat future-proof is to introduce a high degree of flexibility in it. Flexibility in the solution implies it is capable of being used in scenarios that have not been foreseen by its developers at the time of proposal or design, thus allowing it to adapt to new forms of the same or similar threats in the future. Two approaches to achieving this is to either make the defense inherently programmable or allow software to define and set its usage policies. Instances of the former are the general purpose taint tracking accelerators proposed recently [37].

With REST, we take the latter approach wherein our entire defense depends on a hardware primitive, which by itself is not necessarily security-oriented. The where, when, and how of using this primitive is entirely managed by software, thus granting the technique a high degree of flexibility over time. For instance, if the software decides to implement a policy of enforcing more security, this can be easily achieved by planting more tokens and/or by adding an element of randomness as to where tokens are planted. The drawback to this approach, however, is that of privilege, i.e., any attacker operating at the same level as the managing software can potentially manipulate how the tokens are distributed. Hence, we observe again that there is no silver bullet—the effectiveness of any solution can only be measured in terms of contextual and operational trade-offs.

Multipurpose Solutions are Desirable. Defense in depth is an age-old adage in security. It is an acknowledgement of the fact that no single security technique can stop all threats; so system defense is just a question of stacking techniques one over another in orde to raise the bar high enough that exploiting the system presents an unfavorable cost-vs-benefit ratio for the attacker. So, naturally, the more angles of defense a technique presents, the better its defensive value. This is not typical for most previously proposed solutions which are targeted towards one and only one problem. This is a hard quality to impart a solution because often different types of threats do not contain any common feature (although in order to chain the entire attack, one might be a prerequisite for another). So, it is better to do one job well and (reasonably) succeed than attempt many and fail at all. However, when possible, such an avenue could be worth exploring, if it makes practical and economic sense.

With this in mind, we have designed Polyglot to have multiple defensive angles. Preventing runtime code reuse does not require the asymmetric encryption of keys (used to symmetrically encrypt code) in the binary with the device's own unique key. In fact, had that been the only goal, having this feature negatively impacts all aspects of the design's evaluation. However, we considered the addition of that design point reasonable because we considered the consequent cost-benefit trade-off practical. Specifically, we consider the tying of code to a particular hardware instance beneficial despite the small performance, area, and power overhead it incurred. The two points of preventing code reuse and drive-by download style attacks are mutually exclusive and have traditionally required different solutions. However in our case, we could achieve both by introducing a small incremental unit in our design.

Make it Lucrative for All Parties. This point emphasizes the overarching narrative of hardware design in general and reiterates all the points made earlier on a broader scale. It is ultimately the needs of the market that decides the worth of a feature. Hardware and software developers have to work in tandem to deliver features that are demanded by the users to the extent that it makes economic sense. As such, the ecosystem involves multiple players, and it is the job of the designer to come up with a feature that satisfies and balances all their needs. This means that the feature should satisfy the following parties thus. One, the user demands performance and energy-efficiency from the technique. To make matters challenging still, since security is not yet a primary concern among a significant portion of the consumer base, these bounds can be tighter for defensive features than say, usability or purely performance optimizations. Secondly, the software developer demands convenience of feature integration. This is primarily due to the prevalence of legacy code and the high inertia of changing established work flows. This can be achieved in many ways from adding it automatically with compiler passes or making it available through small stand-alone code modules. And lastly, the hardware vendor demands ease and security of deployment. As we discuss at the outset, this means that the technique should satisfy the concerns of effectiveness, longevity, and complexity of implementation. Admittedly, listed as such, balancing all these diverse concerns may seem like a daunting task. My belief, however, is that if we follow the thesis of this dissertation, we will be able to design solutions that meet all of these criteria satisfactorily, thus significantly and positively impacting the user in practice, which is our ultimate goal as computer engineers at the end.

Bibliography

[1] *AddressSanitizer Algorithm*. https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[2] *AddressSanitizer in Hardware*. https://github.com/google/sanitizers/wiki/AddressSanitizerInHardware.

[3] *ARM A64 Instruction Set Architecture for ARMv8-A architecture profile*. "https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf". 2018.

[4] *ARM Cortex A-15 Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438c/DDI0438C_cortex_a15_r2p0_trm.pdf.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code". *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[6] Michael Backes and Stefan Nürnberger. "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing". *23rd USENIX Security Symposium (SEC)*. 2014.

[7] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. "Randomized instruction set emulation to disrupt binary code injection attacks". *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*. 2003.

[8] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M. Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, and Michael Franz. "Hardware Assisted Randomization of Data". *Proceedings of 21st International Symposium of Recent Advances in Intrusion Detection*. RAID. 2018.

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

[10] Sandeep Bhatkar and R. Sekar. “Data Space Randomization”. *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2008.

[11] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. “Efficient Techniques for Comprehensive Protection from Memory Error Exploits”. *Proceedings of USENIX Security (SSYM)*. 2005.

[12] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. “Timely Rerandomization for Mitigating Memory Disclosures”. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. “The Gem5 simulator”. *SIGARCH Computer Architecture News* (2011).

[14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. “Hacking Blind”. *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. 2014.

[15] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. “Prince–a low-latency block cipher for pervasive computing applications” (2012).

[16] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. “On the General Applicability of Instruction-Set Randomization”. *IEEE Trans. Dependable Secur. Comput.* ().

[17] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Leakage-Resilient Layout Randomization for Mobile Devices”. *23rd Annual Network & Distributed System Security Symposium (NDSS)*. 2016.

[18] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. “A Practical Approach for Adaptive Data Structure Layout Randomization”. *European Symposium on Research in Computer Security (ESORICS)*. 2015.

[19] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. “Non-control-data Attacks Are Realistic Threats”. *Proceedings of the 14th conference on USENIX Security Symposium (SSYM)*. 2005.

[20] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. “StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries.” *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2015.

[21] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. “Ropecker: A generic and practical approach for defending against rop attacks”. *In Symposium on Network and Distributed System Security (NDSS)*. 2014.

[22] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. "CHERI JNI: Sinking the Java Security Model into the C". *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[23] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. 2015.

[24] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[25] *Chromium Project: AddressSanitizer*. https://www.chromium.org/developers/testing/addresssanitizer.

[26] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks". *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[27] *Control-flow Enforcement Preview*. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. 2017.

[28] Jonathan Corbet. *An updated guide to* `debugfs`. http://lwn.net/Articles/334546/. May 2009.

[29] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure". *IEEE Symposium on Security and Privacy (SP)*. 2015.

[30] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks". *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[31] *CVE-2014-0160*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. 2014.

[32] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[33] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming". *22nd Annual Network and Distributed System Security Symposium (NDSS)*. 2015.

[34] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the Feasibility of Online Malware Detection with Performance Counters". ISCA '13 (2013).

[35] Solar Designer. *Getting around non-executable stack (and fix)*. http://seclists.org/bugtraq/1997/Aug/63. Aug. 1997.

[36] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. "Hardbound: Architectural Support for Spatial Safety of the C Programming Language". *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008.

[37] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and Andre DeHon. "Architectural Support for Software-Defined Metadata Processing". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. 2015.

[38] Gregory J. Duck and Roland H. C. Yap. "Heap Bounds Protection with Low Fat Pointers". *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 2016.

[39] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. "Stack Bounds Protection with Low Fat Pointers". *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[40] Morris Dworkin. *Recommendations for Block Cipher Modes of Operation: Methods and Techniques*. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. 2001.

[41] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution". *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[42] Exploit Database. *EBD-31574*. Feb. 2014.

[43] *Firefox and Address Sanitizer*. https://developer.mozilla.org/en-US/docs/Mozilla/Testing/Firefox_and_Address_Sanitizer.

[44] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. "Silicon Physical Random Functions". *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. 2002.

[45] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. "Architectural support for low overhead detection of memory violations". *2009 Design, Automation Test in Europe Conference Exhibition*. 2009.

[46] Jason Gionta, William Enck, and Peng Ning. “HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities”. *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2015.

[47] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. “A Case for Unlimited Watchpoints”. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2012.

[48] *Hardware-Assisted Checking Using Silicon Secured Memory (SSM)*. https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html. 2015.

[49] John L. Henning. “SPEC CPU2006 Benchmark Descriptions”. *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17.

[50] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks”. *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[51] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. “Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation”. *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE)*. 2006.

[52] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. May 2011.

[53] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*.

[54] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. “Cyclone: A Safe Dialect of C”. *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (Usenix ATC)*. 2002.

[55] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “SCRAP: Architecture for Signature-based Protection from Code Reuse Attacks”. *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA ’13. 2013.

[56] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. “Countering code-injection attacks with instruction-set randomization”. *ACM Conference on Computer and Communications Security (CCS)*. 2003.

[57] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. “kGuard: Lightweight Kernel Protection against Return-to-user Attacks”. *Proceedings of the* 21^{st} *USENIX Security Symposium (USENIX Sec)*. 2012.

[58] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software”. *Proceedings of the* 22^{nd} *Annual Computer Security Applications Conference (ACSAC)*. 2006.

[59] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. 2014.

[60] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre attacks: exploiting speculative execution”. *arXiv.org* (Jan. 2018).

[61] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios Kemerlis, and Michalis Polychronakis. “Compiler-assisted Code Randomization”. *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[62] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and Andre DeHon. “Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security”. *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS)*. 2013.

[63] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. “SoK: Automated Software Diversity”. *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. 2014.

[64] Kevin P. Lawton. “Bochs: A Portable PC Emulator for Unix/X”. *Linux Journal* 1996.29es (1996), p. 7.

[65] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. “Architecture for Protecting Critical Secrets in Microprocessors”. *Proceedings of the 32nd annual international symposium on Computer Architecture (SP)*. 2005.

[66] *Leon3 Processor*. http://www.gaisler.com/index.php/products/processors/leon3.

[67] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. “Polymorphing Software by Randomizing Data Structure Layout”. *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA ’09. 2009.

[68] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown”. *arXiv.org* (Jan. 2018).

[69] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. “UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages”. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[70] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. “ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks”. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[71] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005.

[72] V. Gayoso Martinez, F. Hernandez Alvarez, L. Hernandez Encinas, and C. Sanchez Avila. “A comparison of the standardized versions of ECIES”. *Information Assurance and Security (IAS), 2010 Sixth International Conference on*. 2010.

[73] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*. 2010.

[74] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. "Flicker: An Execution Infrastructure for Tcb Minimization". *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. 2008.

[75] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

[76] Larry McVoy and Carl Staelin. "Lmbench: Portable Tools for Performance Analysis". *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC)*. 1996.

[77] Matt Miller. *Safely Searching Process Virtual Address Space*. `http://www.nologin.com/Downloads/Papers/egghunt-shellcode.pdf`. Sept. 2004.

[78] *MKLINUXIMG-2.6.36*. `http://www.gaisler.com/index.php/downloads/linux`.

[79] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Watchdog: Hardware for safe and secure manual memory management and full memory safety". *39th International Symposium on Computer Architecture (ISCA)*. 2012.

[80] Santosh Nagarakatte, Milo Martin, and Steve Zdancewic. "WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking". *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2014.

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

[82] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: Type-safe Retrofitting of Legacy Software". *ACM Transactions Programming Language Systems* (2005).

[83] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2007.

[84] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. "TrustZone Explained: Architectural Features and Use Cases". *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 2016.

[85] Gene Novark and Emery D. Berger. "DieHarder: Securing the Heap". *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. 2010.

[86] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches". *CoRR* (2017).

[87] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael B. Abu-Ghazaleh, and Dmitry V. Ponomarev. “Malware-aware processors: A framework for efficient online malware detection”. *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. 2015, pp. 651–661.

[88] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. “ASIST: Architectural Support for Instruction Set Randomization”. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*. 2013.

[89] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. “Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization”. *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 2012.

[90] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing”. *Proceedings of the 22Nd USENIX Conference on Security (SEC)*. 2013.

[91] PaX Team. *PaX address space layout randomization*. 2010.

[92] *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[93] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. “DynaGuard: Armoring Canary-based Protections Against Brute-force Attacks”. *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*. 2015.

[94] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. “Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite”. *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. 2007.

[95] Georgios Portokalidis and Angelos D. Keromytis. “Fast and practical instruction-set randomization for commodity systems”. *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. 2010.

[96] Feng Qin, Shan Lu, and Yuanyuan Zhou. “SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs”. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*. 2005.

[97] Qualcomm Technologies Inc. *Pointer Authentication on ARMv8.3*. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. 2017.

[98] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. “Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications”. *IEEE Symposium on Security and Privacy (SP)*. 2015.

[99] Jeff Seibert, Hamed Okhravi, and Eric Söderström. “Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code”. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[100] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (Usenix ATC)*. 2012.

[101] Hovav Shacham. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”. *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*. 2007.

[102] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase Tracking and Prediction”. *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA ’03. 2003.

[103] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. “High Efficiency Counter Mode Security Architecture via Prediction and Precomputation”. *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA)*. 2005.

[104] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. “Reviving instruction set randomization”. *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. HOST ’17. 2017.

[105] Kanad Sinha and Simha Sethumadhavan. “Practical Memory Safety with REST”. *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. 2018.

[106] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. 2013.

[107] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. “HDFI: Hardware-Assisted Data-Flow Isolation”. *2016 IEEE Symposium on Security and Privacy (SP)*. 2016.

[108] Alexander Sotirov. “Heap Feng Shui in JavaScript”. *Black Hat Europe*. 2007.

[109] Ana Nora Sovarel, David Evans, and Nathanael Paul. “Where’s the FEEB? The effectiveness of instruction set randomization”. *Proceedings of the 14th conference on USENIX Security Symposium (SEC)*. 2005.

[110] National Institute of Standards and Technology. *FIBS 197, Advanced Encryption Standard (AES)*. Tech. rep. Nov. 2001.

[111] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. “Efficient Memory Integrity Verification and Encryption for Secure Processors”. *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2003.

[112] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. “AEGIS: architecture for tamper-evident and tamper-resistant processing”. *Proceedings of the 17th annual international conference on Supercomputing*. 2003.

[113] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure Program Execution via Dynamic Information Flow Tracking". *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. 2004.

[114] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. 2013.

[115] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads". *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[116] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. "Architectural Support for Copy and Tamper Resistant Software". *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2000.

[117] *Trusted Computing Group*. `http://www.trustedcomputinggroup.org`. 2003.

[118] *uClibc-0.9.33.2*. `http://www.uclibc.org`.

[119] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging". *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*. 2007.

[120] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. "CODOMs: Protecting Software with Code-centric Memory Domains". *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. 2014.

[121] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. "High System-Code Security with Low Overhead". *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 2015.

[122] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. "SecPod: a Framework for Virtualization-based Security Systems". *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015.

[123] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. 2015.

[124] Yoav Weiss and Elena Gabriela Barrantes. "Known/Chosen Key Attacks against Software Instruction Set Randomization". *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. 2006.

[125] David Weston and Matt Miller. "Windows 10 Mitigation Improvements". *Black Hat USA*. 2016.

[126] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. 2014.

[127] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. "Improving Cost, Performance, and Security of Memory Encryption and Authentication". *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA)*. 2006.

[128] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and J. Torrellas. "iWatcher: efficient architectural support for software debugging". *Proceedings of 31st Annual International Symposium on Computer Architecture, 2004*. 2004.

Appendix

Algorithms Used to Leak XOR and Transposition Keys

• **XOR.** Since 128 bits covers 4 instructions, we have to accurately guess 4 instructions to figure out the key. Additionally, since XOR does not have a "carry" effect, we can also guess the key piecewise. For instance, we can guess two instructions from one location, and two from another (while adhering to the 128-bit granularity) and combine the two portions to figure out the entire key.

Since every fourth instruction is encrypted with the same key chunk, our approach was to try XOR'ing every combination of instructions 4 words apart, and comparing it with the corresponding value in the plaintext in the same function. For a match, XOR'ing the plaintext and the corresponding cipher yields a possible value of the key. Without randomization, it would be trivial to match pairs. With randomization however, it is more difficult to ascertain which corresponding matching pairs. To overcome this, we use a simple frequency heuristic wherein we choose the key chunks encountered most often. The actual algorithm is presented in Algorithm 1.

• **Bit Transposition.** Transposition was harder to break using simple heuristics than it was for XOR. It becomes especially so if we try to guess the key using wrong key match-pairs. In other words, even if we do get two sets of instructions, encrypted and the corresponding plaintext, we might not get the key if do not match them in the right order. Our basic algorithm is shown below in Algorithm 2.

Although checking validity of a likely key candidate was not difficult for us since we perform a static analysis, practically the same can be achieved by verifying that decrypted words correspond to valid SPARC instructions. Additionally, we fine-tune a few parameters (`num_freq`, for instance) after some amount of experimentation. Note that we did not do any extensive analysis to find the most efficient heuristics.

Algorithm 1 Finding XOR Key

```
1:  procedure FIND_XOR_KEY
2:      for all randomized functions, randfunc, and corresponding function, origfunc do
3:          funclen ← length(origfunc)
4:          origpairs ← {$orig_1$, $orig_2$}, s.t. $orig_1$, $orig_2$ ∈ origfunc, and are 4 words apart
5:          randpairs ← {$rand_1$, $rand_2$}, s.t. $rand_1$, $rand_2$ ∈ funclen instructions of randfunc, and are 4 words apart
6:
7:          for all {$orig_1$, $orig_2$} ∈ origpairs and {$rand_1$, $rand_2$} ∈ randpairs do
8:              if XOR($orig_1, orig_2$) = XOR($rand_1, rand_2$) then
9:                  key_cand ← key_cand+XOR($orig_1, orig_2$)
10:             end if
11:         end for
12:
13:         if check_success() then
14:             $key$ ← 4 most frequent elements in key_chunks
15:             return SUCCESS
16:         end if
17:     end for
18:     return FAIL
19: end procedure
```

Algorithm 2 Finding Transposition Key

```
keys[i] ← [0,31], ∀i ∈ [0,31]

procedure FIND_TRANSPOSITION_KEY
    for all randomized functions, randfunc, and corresponding function, origfunc do
        Generate rand_f such that $rand\_f[val]$ = {inst i} ∀ i that appears $val$ times in randfunc
        Generate orig_f such that $orig\_f[val]$ = {inst i} ∀ i that appears $val$ times in origfunc
        $num\_cands$ ← minimum number of candidate instructions before brute-forcing is attempted
        $min\_freq$ ← minimum frequency of instruction before it is considered for brute-forcing
        if sum(len($rand\_f[i]$) $\geq num\_cand$, where $i \geq min\_freq$ then
            GUESS(orig_f[min_freq:], rand_f[min_freq:])
        end if
    end for
    return FAIL
end procedure

procedure GUESS(setA, setB)
    for all $x \in$ [0,len($setA$)] do
        $inA \leftarrow setA[x]$
        $inB \leftarrow setB[x]$
        for all $i \in [0, 31]$ do
            $bitA \leftarrow (inA\&(1 \ll i)) \gg i$
            for all $j \in [0, 31]$ do
                $bitB \leftarrow (inB\&(1 \ll j)) \gg j$
                if $bitB \neq bitA$ then
                    $keys[i].remove(j)$
                    if len($keys$) = 0 then
                        return FAIL
                    end if
                end if
            end for
        end for
    end for
    if VALIDATE($keys$) then
        return SUCCESS
    end if
    if !CHECK_PARTIAL_VALIDITY(keys) then
        return FAIL
    end if
    if Set elements in $keys$ small then                                ▷ Try brute-forcing
        for all $key\_comb \in keys$ do
            if VALIDATE($key\_comb$) then
                return SUCCESS
            end if
        end for
    end if
    return FAIL                                                         ▷ Try again later
end procedure
```