Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference

Mohit Tiwari Xun Li Hassan M G Wassel Frederic T Chong Timothy Sherwood
Department of Computer Science, University of California, Santa Barbara
{tiwari,xun,hwassel,chong,sherwood}@cs.ucsb.edu

ABSTRACT

High assurance systems such as those found in aircraft controls and the financial industry are often required to handle a mix of tasks where some are niceties (such as the control of media for entertainment, or supporting a remote monitoring interface) while others are absolutely critical (such as the control of safety mechanisms, or maintaining the secrecy of a root key). While special purpose languages, careful code reviews, and automated theorem proving can be used to help mitigate the risk of combining these operations onto a single machine, it is difficult to say if any of these techniques are truly complete because they all assume a simplified model of computation far different from an actual processor implementation both in functionality and timing. In this paper we propose a new method for creating architectures that both a) makes the complete information-flow properties of the machine fully explicit and available to the programmer and b) allows those properties to be verified all the way down to the gate-level implementation the design.

The core of our contribution is a new call-and-return mechanism, *Execution Leases*, that allows regions of execution to be tightly quarantined and their side effects to be tightly bounded. Because information can flow through untrusted program counters, stack pointer or other global processor state, these and other states are *leased* to untrusted environments with an architectural bound on both the time and memory that will be accessible to the untrusted code. We demonstrate through a set of novel micro-architectural modifications that these leases can be enforced precisely enough to form the basis for information-flow bounded function calls, table lookups, and mixed-trust execution. Our novel architecture is a significant improvement in both flexibility and performance over the initial Gate-Level Information Flow Tracking architectures, and we demonstrate the effectiveness of the resulting design through the development of a new language, compiler, ISA, and synthesizable prototype.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*; C.3 [**Special Purpose And Application-Based Systems**]: [Real-time and Embedded Systems]

General Terms

Security, Reliability, Design

Keywords

High assurance systems, Gate Level Information Flow Tracking, Timing channels, Covert channels

1. Introduction

Systems responsible for controlling aircraft, protecting the master secret keys for a bank, or regulating access to extremely sensitive commercial or military information, all demand a level of trust far beyond the norm. Such *High-Assurance Systems* must be secure (to prevent unauthorized disclosure of sensitive information), safe (to prevent unintended events, especially if they can be caused by external agents), real-time (to meet critical deadlines), and fault-tolerant (to guarantee a certain quality of service despite failures, workload, or environmental anomalies) [20]. Creating these systems today is an incredibly expensive operation both in terms of time and in terms of money; even assessing the assurance of the resulting system can cost upwards of $10,000 per line of code [2].

One of the reasons that designing and verifying these systems is so challenging, is that it is difficult to bound how and when the different components of the system interact with one other. One of the strictest properties one might wish to demonstrate is *non-interference*, which in turn can be used to prove that information cannot leak (for security), that untrusted information was not used in the making of critical decisions (for safety and fault tolerance) nor in determining the schedule (real-time). Non-interference requires showing that *sensitive* inputs can never have a measurable effect on an output marked as *non-sensitive*, a task for which traditional microprocessors are very poorly suited. Almost every recent microarchitectural technique is built around the notion of optimizing the common case, an end achieved in large part through the addition of caches, status bits, exceptions, predictors, and other behaviors that modify the state of the machine. The problem is that, if one is protecting a secret or handling untrusted data, every operation performed on that secret will affect those internal states in one way or another. Non-interference requires that those affected internal states are then in no way visible to the other components, including either directly through the ISA, or

indirectly through the resulting differences in behavior or timing. Rather than trying to patch up all the complicated information flow problems presented by traditional processors with software after the fact, we propose: a) that new architectures can be created capable of strongly containing information, b) that such architectures will allow software to easily verified for properties even as strict as non-interference, and that c) if built in a specific way, such properties can be verified all the way down to the gates of the processor leaving no room for covert or timing channels due to the processor's RTL implementation. On such a machine it is possible to *tightly quarantine* computations which are either secret (to prevent the loss of those secrets) or untrusted (to prevent those untrusted operations from effecting critical state)

We introduce *Execution Leases*, an architectural mechanism that makes information flows explicit to the programmer, including timing, covert and implicit flows through control/architectural state. The basic idea behind a lease is that control of a portion of the machine is given over to an untrusted entity for a fixed amount of time and within a fixed range of addresses. After the lease expires, control is yanked back to the trusted code and any remnants of the untrusted actions are purged from the critical machine state such as the PC (registers and main memory are not part of the critical machine state and retain their values and their security labels even after a lease expires). The hard part is performing this operation with reasonable overheads and in a way that can be demonstrated to be correct through inspection of the gate-level implementation alone. Because of the relative freedom provided by Execution Leases (as opposed to the past approach of ensuring there is never *any* way for untrusted or sensitive code to effect the program counter), the resulting code can be a 100x faster in some cases, several factors smaller, and far easier to program. Specifically, our contributions include:

- The introduction of Execution Leases, both as a programming model in the abstract and as implemented by a specific ISA for high assurance systems.
- New methods for verifiable architectural information-containment by design, including a description of the various microarchitecture modifications needed to bound information flows in our processor.
- The evaluation of a complete Execution Leases implementation, including a new Lease-based ISA, a small language and compiler that target this ISA, a fully synthesizable prototype, a complete gate-level information flow analysis of the final design, and results from experiments with several hand-written applications.

We begin by more precisely defining the concept of non-interference, the motivation behind its application by the high assurance community, and the several other projects most closely related to Execution Leases (in Section 2). The specifics of the architecture, along with details of its implementation and application are presented in Sections 3 and 4. Finally we describe experimental results in Section 5 and conclude with Section 6.

2. Motivation and Related Work

The strict demands on the design of high assurance systems are a direct function of the cost of failure. As an example, Boeing plans to use a single physical network for both aircraft control data and passenger network data in its new 787 [9], and keeping these two *very strictly* separated is obviously critical. The task of ensuring this separation falls to custom software written to implement ARINC, an international standard that specifies the communication protocols between different entities in avionic systems, including the bus protocols between processors and the interfaces between software modules and the separation kernel. An ARINC bus uses two different packet formats, one for critical data and another for non-critical file transfers (such as movies), and, as a standard, can be implemented differently by many vendors. From these different software components Integrated Modular Avionics (IMA) systems are assembled, and one of the most critical tasks in ensuring their safety is showing that different components executing on the same hardware platform are strictly partitioned and cannot interfere with one another.

To help vendors and systems integrators discuss the level of trust that can be afforded to a particular system, the common criterion establishes a set of 7 Evaluation Assurance Levels (from EAL1 to EAL7). An EAL7 means the product has been formally designed, tested, and verified to be provably sound from the ground up. While no operating system has ever been rated EAL7, the Integrity RTOS [1] is one of the closest at EAL6+. Even getting through the evaluation to EAL6+ required over $10K per line of code, totaling millions of dollars over 10 years, and in the end it is not clear that the end product will ever even make it to EAL7 because of the difficulties in formally proving the most critical aspects of the design. One of the primary difficulties in getting software verified at these levels is that modern machines are simply not built with the idea of information containment in mind.

While the above examples discuss the problem of interference between various software components on the same processor from the view point of preserving *integrity*, exactly the same problem occurs in systems concerned with *secrecy*. If a secret such as a private key is used in performing an operation, an attacker may be able to reverse engineer the key through direct timing observations[12], cache interference[23], even through the state of the branch predictor [4]. Once an attack has been identified and publicized, effective countermeasures can be deployed, e.g. randomizing the replacement policy for the cache[16], but this constant cycle of attack-and-respond is unsatisfying when the cost of leaking some data is extraordinarily high (for example the root private key for a bank, or a military authorization code).

To address both secrecy and integrity, one of the strongest properties one can show about a system is *non-interference*. Non-interference means that, if inputs and outputs are considered either "High" (e.g. secret) or "Low" (e.g. unclassified), then a change in a "High" input can never be observed or inferred from changes in the "Low" output. In other words, "High" data should never leak to "Low". This property can then be used to ensure that no secret input ever affects an unclassified output, or conversely that no untrusted input ever affects a high-integrity output. To achieve non-interference, all sensitive data must be carefully guarded to prevent even an adversary from observing or inferring its contents. While this basic assumption underlies countless trusted systems, preventing an adversary from inferring *any* information is very difficult to ensure in practice and even harder to verify formally. Even the possibility of a tiny discrepancy in timing, a moment of unplanned resource contention, or a single loose memory access is enough to lose non-interference and open up a host of cryptographic attacks or bring the integrity of the result into question. Ensuring non-interference means shutting down all channels of communication from "High" down

to "Low". While non-interference is a very strong policy, it is very useful when the integrity of certain applications is paramount (such as aircraft controls). Further, even though strict non-interference does not apply directly to secrecy through encryption, the ability to demonstrate the absence of interference except for the encrypted text can be useful in creating software data diodes and red-black systems.

2.1 Related Work

Information flow policies have been specified and enforced at many levels in the computing hierarchy. Programming language based techniques use sophisticated type systems to represent security levels and to enforce information flow policies statically, and even provide several alternative models in addition to strict non-interference [35, 27, 22, 24]. While some timing channels can be eliminated by executing both sensitive conditional paths atomically or executing dummy instructions [27], PL-only techniques use general purpose processors but assume the information flow within the processor behaves in a way that is far simpler than real implementations dictate (often ignoring caches, arbitration, stalls, etc.). Execution Leases can be shown to be well behaved (containing the flow of information) all the way down to the gate-level implementation, providing a foundation for software schemes to further build upon.

Information flow can also be controlled at the granularity of operating system abstractions. Commercial embedded systems use separation kernels like Integrity [1] to isolate software modules with high assurance. On the other hand, projects such as HiStar [37], Flume [14], and Laminar [25] apply distributed information flow control (DIFC) for more general purpose operating systems and provide finer control over information flows. Such OS level mechanisms are subject to covert channels that may occur through sparsely documented hardware features such as FPU flags [28], or through timing channels inherent in the underlying hardware. Karger et al, in a retrospective of the VAX VMM security kernel [10] highlight the challenges of an OS or hypervisor-level approach mentioning specifically that covert channel analyses "were done on an informal basis by engineers by closely studying system design" and that "timing channels proved a much more serious problem [...] because many of them were inherent in the underlying hardware". Their solution for timing channels was to "fuzz" potential sources of accurate clock information so as to lower the bandwidth of those channels. There have also been recent efforts to formally prove non-interference properties of operating systems [13, 19], that can prove absence of covert channels through kernel data structures. However, these approaches do not handle hardware timing channels. Our work provides mechanisms to close such loopholes because execution leases are information tight with respect to both covert and timing channels (and can be shown so with GLIFT[31]).

The general idea of creating security-enhanced architectures has been around for quite a long time. Encryption and memory partitioning can be used to run trusted software on an untrusted host where even the operating system and main memory are not trusted [29]. Ad-hoc trusted hardware designs such as XOM have been focused on preventing software piracy [18, 36]. It is difficult, however, to prove that incorrect information flows do not exist in these designs and their final implementations. Intel's tag-based architecture takes a different approach to security, adding hardware capable of enforcing arbitrary policies over data structures through the use of custom call-back mechanisms [17]. This idea of tag-based tracking at the architecture or ISA levels has seen a recent resurgence. DIFT [30], Minos [6], Rifle [33], Raksha [8], FlexiTaint [34], Log-Based Lifeguards [26] have all proposed special architectures to track flow of data through the registers and memory, while Dytan [5], Taintcheck [3] and others explore efficient binary instrumentation-based methods of doing the same. All the above ISA-level proposals, while capable of exposing many vulnerabilities, track only explicit information flows and do not handle implicit or timing channels.

In fact, to track all information flows in a precise and sound manner, we would need to mark *all* of the processor and memory state that *could have been touched by any possible path of execution* that resulted from every branch. Failure to do so creates covert channels that have been notoriously difficult to analyze in systems to date [32, 15, 21, 11, 10]. Past techniques suffer from the fact that they operate exclusively at the ISA level or above – they do not capture those implementation details (such as bus arbitration or forwarding logic) that can lead to timing channels. Execution Leases, in comparison, is a new concept in high assurance architectures that allow all information flows to be explicitly tracked by providing direct hardware support for execution that is bounded in both space and time.

2.2 Gate-Level Information Tracking

Because we want our implementation and analysis of Execution Leases to be demonstrably sound from the gates up, we perform this work under the framework of Gate-Level Information Flow tracking (GLIFT [31]). To understand the main idea behind GLIFT, let us consider a single two-input multiplexer. Multiplexers are both complete (you can create any logic function from MUXes) and exceedingly useful in real designs. Consider a MUX with three binary inputs (A, B, select S) and an output O. For the purpose of discussion, let us assume that this MUX is our entire system, and that the inputs to this MUX can come from either tainted or untainted sources. Let us further assume that those inputs are marked with a bit (A_t, B_t, and S_t respectively) such that a 1 indicates that the data is tainted. The basic problem of gate-level information flow tracking is to determine, given some input (A, B, and S) and their corresponding tainted bits (A_t, B_t, and S_t) whether or not the output O should be marked as tainted or not (which is then added as an extra output of the function O_t). A MUX here is a good example because, intuitively speaking, the result should be tainted if either a tainted input is selected or the select bit is tainted. Consider the case where $A = 0, A_t = 1, B = 1, B_t = 0$, and $S_t = 0$. If $S = 0$ (meaning to select A), O_t should be 1, while if $S = 1$, O_t should be 0. In fact this intuitive idea can be formalized quite nicely with the following property: the output of a logical function should only be marked as tainted if and only if some set of tainted inputs actually had an opportunity to affect the output. In other words, if it is possible to affect a change in the output through any changes in any of the tainted inputs, the output should be marked as tainted.

Gate-Level Information Flow tracking in and of itself does not ensure that critical or untrusted data do not spread across the whole machine, rather it only ensures that as critical or untrusted data spread across the machine it will always be properly tracked. It is certainly possible to implement a full RISC machine in GLIFT logic, but as soon as the PC is marked as tainted, the entire machine will end up being tainted. Because that untrusted PC could jump to any location, the code that is executing is derived

from untrusted data. The challenge is to create an architecture capable of containing the flow of information so tightly that, by looking only at the gate-level implementation it is apparent that it cannot leak, while simultaneously retaining enough flexibility to be efficiently programmed to a variety of uses. In [31], a very simple architecture is used to show that this is indeed possible in principle by removing all branch instructions (replacing them with predicates) and ensuring that there is no possible way for the program counter to *ever* be effected by any tainted data. This *severely* restricts the usefulness of the architecture – to even lookup an entry in a simple table of size n takes (n) steps instead of the (1) that it would with a simple indirect load. In this paper we show how Execution Leases allow this and many other restrictions of this original architecture to be lifted, resulting in a far easier to program and faster system.

3. Architecture

To understand the reasons behind Execution Leases, we begin by describing the many ways in which information can leak in a traditional processor, and the lengths that the original GLIFT processor went to in order to prevent them.

3.1 The Problem with Overprotecting Critical State

Constructing an efficient architecture that can strongly contain the flow of information, yet still maintains a good level of programmability is difficult, and the philosophy for dealing with this problem in the original GLIFT work was simply to ensure that critical machine state could never become tainted. While this sounds straight forward, it is quite a bit harder than it sounds. It means that the architecture has to be constructed in such a way that it is *impossible for any data* in the system (which could then turn out to be tainted) to *ever* effect the program counter, the instruction memory, or the address of a store. If any of these were to be tainted, the entire state of the machine would quickly (in one or two cycles) end up marked as tainted, and there would be no way to undo that damage.

As an example: the original GLIFT architecture ensured that it was impossible for the program counter to ever be influenced by the the result of some computation (and thus risk being tainted). This, of course, means no conditional jumps of any sort, and in fact ensures that programs will always execute a set fixed number of instructions at every invocation. While the machine would no longer be Turing-complete, in many cases the complete lack of conditional jumps could be compensated for by predication. Because predication transforms control dependencies into data dependencies, almost all of the instructions in the original GLIFT-enhanced architecture could be executed conditionally without ever effecting the program counter (with the obvious exception of jumps).

Likewise, the architecture had to prevent the execution of indirect loads and stores. Consider the information flow in the statement $M[x] = 1$. In this statement, there is clearly a flow of information from x to $M[x]$, but there is also a much more subtle *implicit* flow of information from x to $M[y]$ where $x \neq y$. Why? Because by observing that $M[y] \neq 1$, we have now gained some information about x. If a store is to be executed and the target address of the store is tainted, information flows from that store the *every single piece of memory in the system* (in other words every possible value of y). If we think about the flow of information at the gate level, this becomes very clear. The tainted bits of the

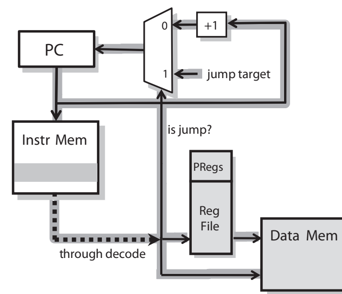

Figure 1: Figure shows the basic GLIFT processor running tainted code. The taint (in gray) spreads to the entire system state and makes it practically useless to track information flows.

address flow into the memory decoder, and as a result all of bit-lines are tainted. When the write actually happens, we have to assume that any of those bit-lines could have been active and thus the write could have happened to any of the possible memory addresses. To deal with this problem, the original GLIFT architecture prevented any indirect loads and stores, instead enforcing that all of the loads and store used an address that was a constant offset from some untaintable counters (kept untaintable in much the same way as the PC was above).

The final, and perhaps most obvious step to keeping the core architectural state from becoming tainted in the original GLIFT architecture is to never allow the execution of *tainted code*. Tainted code, will always end up tainting the PC, the load and store address, and any other state effected by the execution of that code (i.e. everything), as shown in Figure 1.

The ramifications of this philosophy of never allowing any of the processor critical processor state to become tainted are enormous. For example to do a table lookup (a very common operation in AES and many other crypto algorithms), due to a lack of indirect loads, the original GLIFT architecture would have to loop over all of the entries and predicate out all of the loads that were not the one specific index that was to be looked up – a slow and code intensive prospect. It also lead to an inability to have functions, and to bound the effect of an execution of tainted code[1]. The approach we take in this paper is far less constrained, allowing all of this critical state to become tainted over bounded periods of time, while always keeping the *minimal trustworthy control* necessary to return the machine to a fully untainted state.

3.2 Bounding and Cleaning up Tainted State with Execution Leases

To understand how an Execution Lease helps to solve this problem, let us first consider the execution of arbitrary tainted code. This case, where the bits of the actual instruction are tainted, is the most difficult to bound. An untainted function `foo` (e.g. some trusted function), wishes to call a tainted function `bar` (e.g. some arbitrary code). On a traditional machine, this could be implemented with a call and return. The problem is that once the PC jumps to tainted code, everything that code does is tainted, even the eventual return instruction.

[1]If you are considering a confidentiality policy where *secrets* are tainted rather than untrusted data, this ability would be particularly useful because it means that code itself could be secret and there would be no way to learn either what the code is or the results of its computations without observing bits marked as tainted (secret)

Instead we need a way to jump into that tainted code such that a) we can get back to `foo` without learning *anything* about what happened inside `bar`, and b) we need a way to bound *all* the state that can be changed by `bar` so that `foo` can't learn about `bar` by observing things that `bar` did not do (i.e. bound the implicit flows). Because confidentiality and integrity are different forms of the same problem we can phrase the exact same property by just changing the way "taint" is interpreted after the analysis is complete.

The idea behind an Execution Lease is to grant access to a limited amount of state of the machine (including the PC and a portion of the memory) for a fixed and predetermined amount of time in such a way that i) enforcement of the lease can *never* by affected by tainted data, ii) the *critical* tainted state (e.g. the PC) can be scrubbed leaving no residue of tainted data behind, and iii) that it is clear through a *gate-level analysis* of the flow of information that properties (i) and (ii) hold (e.g. it does not depend on some property of the software or some semantics of some state to show that (i) and (ii) hold). We implement these execution leases with special instructions `settimer` and `setbounds` that enforce a bound on the number of instructions that can be executed before control is restored back to the caller and a bound on the accessible memory region respectively. However, to see why and how these semantics keep provably tight control over the flow information we need to explain the implementation of the mechanisms behind these semantics.

4. Mechanism

To make sure that the called context (the leasee) does not interfere with the calling context (the leaser), an Execution Lease must enforce a bound on the control flow of the leasee. This ensures that control is returned to the leaser in a manner that is in no way dependent on the leasee. In contrast, during a typical function call, the callee determines when (and if) to return to the caller. Additionally, the leaser must enforce a bound on the address space accessible to the leasee to prevent information from being written explicitly throughout the entirety of the machine. Finally, we need to ensure that both control and address bounding can be performed *without ever making an architectural decision based on the taint values*. This is a subtle but very important point. If we use the taint values in determining whether to "admit" or "deny" a particular action, the fact that the status of that action (admit/deny) is visible to the architecture implies that a dangerous flow of information has taken place. In such a case it may be possible to, for example, try writing to particular addresses to see if those writes are permissible, thus learning something about the values written there. In a very real sense, if the architecture is not separated from the GLIFT logic, the GLIFT logic becomes *part of* the architecture logic, and would then be subject to same potential for covert channels and implicit flows as any other architecture logic. Later in this section, we show how this tangling of GLIFT logic and architecture could happen when propagating information flow for an untrusted store instruction. Instead, we need to build an architecture that handles space and time isolation both cleanly (so we can see it to be true at the gate level) and inherently (to avoid the tangling data and taint bits).

4.1 Inherent Enforcement of Time-Bounds

Instead of a call-and-return, we can ensure that control will be restored to the leaser context using a timer. In essence, one leases the program counter out to the leasee

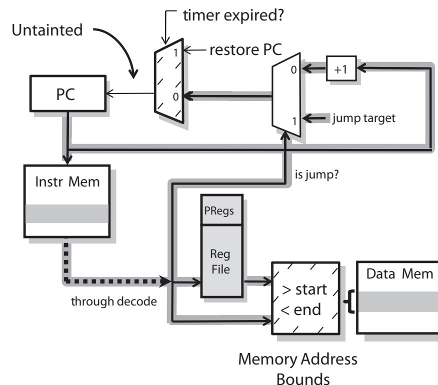

Figure 2: Execution Lease Architecture: Lease logic (dashed) bounds tainted programs in both time and space, and prevents the entire system state from becoming tainted. PC is restored to an untainted restorePC value when an untainted timer expires. Lease logic is also used to bound the memory regions the tainted code can access.

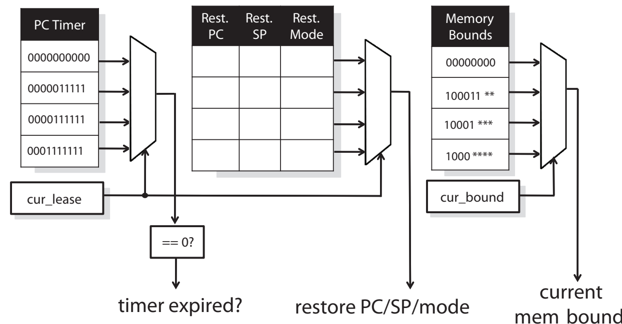

Figure 3: Figure shows the information that has to be stored as part of a stack of successively nested execution leases. Note that a restore SP (here, the cur_lease and cur_bound registers are the stack pointers) is stored with each stack because we do not want to compute the next SP from the current (possibly tainted) SP when a lease expires.

(which may or may not be tainted and where that taint might either indicate secret or untrusted code) for a fixed amount of time. Once the timer expires, control is automatically restored back to a return PC value that was provided by the leaser when it invoked the lease. Figure 2 shows the Lease architecture, and a scenario where untainted code leases the CPU to some tainted code. The timer value itself and the restore PC are untainted, and when the timer expires, a MUX is used to reset the PC to the restore PC. Correspondingly, the GLIFT-logic observes that the MUX output is dependent solely on untainted values (i.e the old tainted PC has no effect), and marks the PC as untainted. Of course nothing is ever this simple, and here the complexity lies in the fact that we need to support *multiple nested leases* to support multi-level procedure calls.

The need for multiple nested leases naturally suggests maintaining a stack of lease records that stores the time the lease is active for and the PC value that the control must return to when the lease expires. We have to implement a stack that stores these attributes, but where the information flow containment is inherent in its gate-level implementation.

With each lease entry in the stack, its leaser's location on the stack is also recorded as part of the lease entry (*restoreSP* in Figure 3). The *restoreSP* thus carries the taint of the leaser, and this allows the Lease-CPU to pop leases by setting the *cur_lease* register to its *restoreSP* value without having to compute the next *cur_lease* value

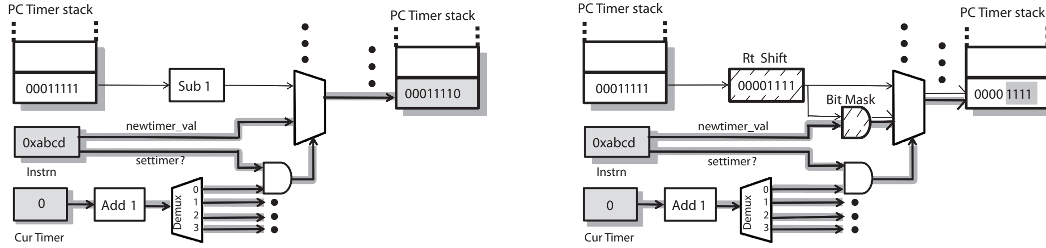

Figure 4: Preventing taint explosion through timers: Figure shows two possible timer implementations in a situation where some tainted code is executing under time bounds set by untainted code. On the left, using an intuitive timer implementation, the taint (gray lines) spreads through the logic and marks the previously untainted timer value as tainted (shaded gray). However, the figure on the right shows that by using a bit-mapped timer encoding and by masking off the leading 0-valued bits from the tainted new timer value, we can ensure that the tainted select input to the MUX can only choose between two identical untainted values for the leading 0-valued bits. This makes it explicit at the gate level that the tainted new timer cannot affect the existing lease timer bounds.

from its current value following the usual stack semantics. If we always use the current value of *cur_lease* to compute the next, once the *cur_lease* register is tainted, it will prevent itself from ever being reset to untainted.

To implement successively nested timers, we encode the timers as a bit-vector where each bit represents a minimum time unit. For instance, a timer of value 00...0111 will execute for three time units. Decrementing these timers then requires shifting the register to the right once every time unit with a 0 entered at the MSB. Nesting of successive timers is enforced at the bit level by using the 0s in the right-shifted current timer value as the prefix of the next timer. Figure 4 (right side) shows how this mechanism provides gate-level guarantees as opposed to an intuitive scheme that used subtractors to decrement the timers (left side in Figure 4). Using the intuitive scheme, an untrusted select input to a MUX decides the next timer value from among the decremented current timer or a new timer value provided by the `settimer` instruction. Even though the decremented timer is trusted, its bit-values could differ from the new, tainted timer value, and because the select itself is tainted, GLIFT logic will mark the MUX output as tainted. As a result, even though we can manually observe that the intuitive implementation is functionally correct, its semantics are not *inherent* in its gate and bit-level implementation. Using our scheme, masking off the trusted, leading bits (0s) ensures that a tainted select chooses between two trusted 0s for the leading bits of the next timer value. As a result, the 0s in a timer value can be shown to increase monotonically until it expires completely. On each cycle, the processor logic detects if the current timer has reached 0 (expired) and if it has, the processor's PC is assigned the current restore PC from the stack.

The timers are bit-encoded in such a way that a few very small sizes are supported in addition to the largest function being covered. In general, since leases require the timing behavior of functions to be specified statically, there is going to be a correlation between the timer encoding and the execution time overhead as compared to a general-purpose version of the program, and as our application suite grows, developing encodings with a wide range will become more important. Since very small leases are often used for small functions and indexing into arrays, in our prototype, we chose to assign two bits each for time granularities of 4 and 32 instructions, and four bits each for 256, 2K, 4K, and 8K instructions. This simple encoding allows lease durations from 4 to 58440 instructions, and is sufficient to cover our application suite.

Finally, we note that the stack of timers come into play only when at least one lease has been set. The processor begins execution in GLIFT mode in a base context that has no corresponding timer, which allows for trusted programs that execute in never-ending loops. We expect that the processor will begin execution in trusted mode in this base context, and because of this the *first lease entry on the PC stack* is expected to always be trusted. We have already discussed mechanisms used to implicitly reset the taint values for important processor state like the PC, (and thus the Instruction word) and the *cur_lease* register. Now we discuss our mechanisms to precisely enforce memory bounds for loads and stores, and the reason behind a power-of-2 aligned memory bounds field.

4.2 Inherent Enforcement of Memory Accessibility

Consider a scheme for enforcing memory bounds that allows a store to go through to memory only if it is within the specified bounds, and some tainted code executing a store instruction. One intuitive option to build such a memory controller would be to use comparators to check if the store address is within bounds, and forward the store instruction to memory only if it is. Figure 5 (left) shows how GLIFT logic will propagate the taint through such a bounds-enforcing logic. Since the address itself as well as the memory's chip-enable is tainted, GLIFT logic for the memory decoder marks *all* the wordlines as tainted.

Instead, in our architecture (right side of Figure 5), memory bounds are stored in ternary format where trailing "*"s represent the desired memory bounds (for e.g. setting bounds register to 10** enforces bounds from 1000 to 1011). A memory controller then composes the address that is actually sent to memory by taking the high bits that are set to either 1 or 0 from the memory bounds register and concatenating the lower bits from the incoming address generated by (potentially untrusted) code. Through such a concatenation, the address sent to memory will always be within the bounds, and the isolation is handled cleanly. Further, this concatenation will create a new address that is partially untainted (the bits that came from the untainted bounds) and only partially tainted (the remaining lower bits extracted from the tainted address). By sending this address and its taint simply to the GLIFT-generated shadow memory decoder, the shadow AND-gates inside the decoder will automatically taint only the address range indicated by the tainted "*" bits. Thus, information flow containment through leases is made *inherent* at the bit level.

To ensure that a `setbound` instruction creates successively nested memory bounds, the bounds are stored as a combination of address and its mask and are thus restricted to be power-of-2 aligned. Each successive bound

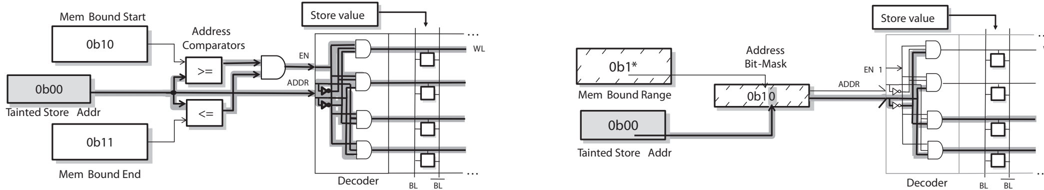

Figure 5: Figure shows (on the left) the problem of implementing memory bounds in a naive fashion, where a tainted address causes all the Word Lines and thus the entire memory to be marked as tainted. On the right, with a bit-masked memory address bound, only the currently accessible memory range is marked as tainted by the GLIFT logic.

is composed using the current bounds' most significant unmasked bits and concatenating the remaining bits from the setbound instruction. In the next section, we show how memory bounds are used in our benchmark programs for function calls as well as for making protected indirect memory accesses. Since these happen independently, we provide independent stacks for the PC and memory bounds (i.e. memory and PC bounds have independent timers). Further, we realized that allowing for two concurrent memory bounds makes it convenient for programs to share memory though global memory regions while also working in their local frames. As a result, we have *two* stacks that enforce two concurrent memory bounds in addition to the stack of PC timers.

4.3 Executing General Purpose Code

We can use the gate-level timing and memory bound guarantees that leases offer to execute general purpose code, specifically conditional jumps and indirect memory accesses, within a lease. For the leasee to be able to use general-purpose instructions, the leaser must set the *Mode* bit for the lease, indicating a general-purpose lease. The mode in which the current lease is executing (GLIFT or General Purpose) forms part of the current context (as shown in Figure 2). The leaser's mode is also stored along with each entry in the PC stack so that the *current_mode* register can be restored to a trusted leaser's mode when a lease expires. An interesting feature of our architecture is that once a lease is set in general purpose mode, no further leases can be set until the lease expires. Leases can only be set in GLIFT mode to ensure that the fixed size lease stack cannot be used to leak information covertly, while within a general purpose lease, conventional call-and-return semantics can be used to implement functions.

Conditional jumps help performance because they can allow the lease time to be limited to the maximum of the two conditional paths at an $if-else$ branch instead of executing both sides of the branch. Similarly, indirect memory accesses allow programs to index into arrays arbitrarily without having to iterate over the entire array and predicating out the desired index. In the next section, we present the Lease ISA and how it can be used to implement a high level language.

5. Evaluation

Now that we have described the basic microarchitectural structures in our Execution Lease prototype, we provide more details of how these structures are exposed to the programmer and compiler. To demonstrate that such an approach can lead to a correct and relatively easy to program secure microcontroller, we have built a fully synthesizable prototype instantiated on an FPGA and a compiler that translates high-level constructs like functions, loops, and array accesses into machine code with lease instructions, jumps and indirect memory accesses. In this section we present performance and area results for this prototype and describe several of the more important features of the design through code examples and comparisons to the original GLIFT work.

5.1 A New ISA for Execution Leases

The original GLIFT architecture uses predication to prevent tainted data from ever affecting the PC, while a special `countjump` instruction allowed a fixed number of unconditional jumps to support fixed-length loops, and special `load-looprel` and `store-looprel` instructions allowed programs to access a fixed range of memory addresses in the loop (by using an immediate value for the base address).

In contrast, our Execution Lease ISA allows the caller of the lease to set explicit bounds on the range of memory addresses that the callee is allowed to access (using `setmembound-hi` or `setmembound-lo`), and the time the callee will execute for (using `settimer`). The `setbounds` instructions set the address bounds for a given time duration and along a given power-of-2 aligned boundary, while the `settimer` instruction sets a timer, the general-purpose mode, and automatically stores a PC to be restored when the lease expires.

Function calls in the new assembly: We have chosen to implicitly record the third instruction after `settimer` (i.e. $PC+3$) as the restore-PC, allowing for one instruction in the middle for an unconditional `jump` to the callee and another instruction that spins in an infinite loop to wait out any remaining time on the lease if the callee finishes early. The address of this infinite-loop instruction is suggested as a return address to the callee as part of the calling convention. Note that even if the callee disregards the return PC and is still executing some code when the lease timer expires, the PC will be yanked back to the restore PC that was recorded when the lease timer was set. The return PC is suggested so that if a lease duration is longer than required, the callee need not be concerned with waiting out their leases or completing the lease precisely without spilling over into others' code. If the time is insufficient, leases will still ensure that the effects of an unusual code-path do not propagate outside the current time and space bounds.

A function call looks like the snippet of assembly code in Figure 6 (that calls an I^2C bus initialization function).

Accelerated array accesses: The `setbounds` and `settimer` instructions are also used to enable indirect memory accesses (used for accessing array elements). Since the GLIFT ISA allowed only direct memory addressing, indexing into an array required using a loop that iterated over the entire array. The chosen address was accessed by using a predicated load (or store) that is set to True at the desired index.

For example, consider a code snippet of the `SubBytes` function for an implementation of the AES [7] encryption algorithm in the original ISA (Figure 7). The function substitutes the value in the `state` matrix with values in the

```
#[Pred] Instruction Operands
[1] setboundg    1000000**, 72   #bounds for global memory for 64 instructions.
[1] ...                          #copy args, ret-addr to callee function.
[1] setboundl    1010100***, 72  #bounds for local stack frame.
[1] settimer     1, 72           #timer := 72. mode := 1 (general purpose).
                                 #restorePC := PC + 3
[1] jmp          i2c_init        #jump to callee function.
label13:                         #function returns here: an infinite loop
[1] jmp          label13         #to wait out remaining lease time.
[1] load-direct  r0, 0x29f       #arrive here when lease expires. mode <= 0.
```

Figure 6: Figure shows the assembly instructions generated to implement a lease called by a programmer in the high-level language. The `setboundg` *and* `setboundl` *instructions set the address bounds for all subsequent memory accesses for the next 72 instructions. The* `settimer` *instruction then initializes the next lease with a mode (general-purpose or GLIFT), a timer and a restore PC, and the following unconditional jump sets the PC to that of the callee function. The PC is expected to return to* `jmp label3`*, where it will spin until it is restored to the* `load-direct` *instruction once the PC timer has counted 72 instructions. Note that the actual function required 42 instructions, and the corresponding local and global timers would be 43 and 50 respectively. Using our bit-encoding of the timers, the timers are set to 72 instructions.*

Mode	Instruction	Information Flow Properties
GLIFT only	setTimer	The lease caller's taint value is assigned to the callee's lease state (like timers, restore PC, restore SP etc). Since the cpu starts off with trusted code, the first timer on the stack is always trusted (i.e untainted).
	setBound g/l	
General Purpose only	jump If RegZero	If jump target or condition Reg is tainted, the PC is marked tainted (following which, all registers and the entire leased memory becomes tainted too).
	jump To RegValue	
Both Modes	load Immediate	The destination register gets the taint value of the PC
	load/store Direct	The destination gets the taint value of the PC or - d with that of the source memory address.
	load PC To Reg	Taint value of PC is assigned to destination Reg.
	load/store Indirect -global	A tainted memory address will mark the entire leased memory as tainted.
	load/store Indirect -local	
	countJump	The PC gets the taint value of the instruction word.
	directJump	
	and, or, not, xor, shl, shr, add, sub, cmpeq, cmplt	Destination reg untrusted if either operand is untrusted. For AND and OR, the trust value is tracked precisely

Figure 8: An overview of the ISA of our prototype architecture, and the information flow tracking policies that are extracted from the actual logic level implementation.

`SBox`. The code in Figure 7 loads the value in the `state` matrix (which in this example is stored at address starting at 0x100) and every element serves as an index to the `SBox` and is substituted by the value in the `SBox` (which is stored at address 0x300). The `SBox` is a 256 entry table, correspondingly the `countjump` instruction 0x05 loops back 255 times just to read a single value from the SBox table [31].

In Lease ISA, the `setbound` instructions set a memory access bound for a limited amount of time. In Figure 7, the lease lasts 1 instruction which allows the program to perform a bounded indirect memory access (as shown in an unoptimized snippet from AES in Figure 7). In fact our compiler conservatively inserts bounds before every memory access, but merges adjacent leases of `setbounds` that have the same address range and creates a longer lasting lease.

Executing General Purpose Code Safely:Leases allow us to execute *conditional jump* instructions safely. By enforcing a tight time and memory bound, a lease ensures that there is no untracked side-effect of the general purpose code. As mentioned in Section 3, general purpose code cannot set any further leases. Lease instructions (that set timers and bounds), if executed in general purpose mode, will not commit and will be equivalent to a no-op. In this mode, functions can use the conventional memory-based stack to implement function calls, and the ISA includes an instruction to load the current PC into a register that is used to compute return addresses for callee functions. Finally, the instruction set includes the usual 2-operand single-destination ALU-ops to execute arithmetic, shift, and compare instructions. Table 8 shows the complete list of instructions supported by our Execution Lease ISA.

5.2 A Prototype Processor that implements Execution Leases

We have built a prototype microcontroller that implements all the mechanisms described in Section 3 to support execution leases. The Lease CPU is a 5-cycle, in-order, unpipelined processor with 64Kb of Instruction and Data Memory, 8 general purpose registers and 8 registers to store the loop counters (that count down the number of iterations for `countjmp` instructions).

Most importantly, the CPU maintains three independent lease stacks (one each for the PC, global and local memory bounds) with each stack allowing upto 8 nested leases. Each lease stack includes registers to store the timers, the return stack pointers for the entry, the current stack pointer, and the restore PC and restore mode for the PC stack, and memory bounds in ternary format for the global and local memory stacks.

The Lease CPU is implemented in Verilog as a structural composition of gates and instantiations along with registers for processor state and block RAMs for Instruction and Data Memory. This structural composition of logic allows us to automatically extract the gate-level taint tracking logic for the entire processor by shadowing all registers and wires, and connecting together the shadow gates and modules in the same manner as their original processor counterparts, resulting in a full shadow machine that operates on taint bits instead of data.

We have used Altera's QuartusII v8.0 to synthesize the Glift-Lease CPU onto a Stratix II FPGA with synthesis settings that balance both area and timing. These area and timing numbers are then compared against the basic Glift CPU presented in [31], and with Altera's Nios RISC processor. We chose Nios-standard core as a point of comparison as it has a simple ISA, is reasonably well-optimized, and is targeted to the same family of FPGAs.

Figure 9 shows the area and timing results from synthesizing all the processors under test. This includes the Nios processor as a general purpose baseline, the basic Glift processor and its version augmented with shadow logic as the Glift baseline, and the Lease CPU (with and without shadow logic) as the processors under evaluation. The left Y-axis shows the area in number of ALUTs (black bar) while the maximum operating frequency values are presented on the right Y axis (gray bar). One important result is that in absolute terms, all these processors are very small in size. Even on the outdated Stratix II EP2S15F67263 FPGA that was used for evaluation, the smallest Nios processor required only 5% of the combinational ALUT resources, while the largest Lease CPU required 35% of the ALUTs. The

C code	GLIFT - Base Asm	Lease Asm
state[i] = sbox[state[i]]; /*sbox: int [256] */	0x00 [1] load - looprel R0 := [0x100 + C0] # R0 = state[i] 0x01 [1] init - counter C1 := 0 # start the loop. j = 0 0x02 [1] cmpeq P1 := C1, R0 # if (j == R0) 0x03 [P1] load - looprel R1 := [0x300 + C1] # R1 = SBox [j] 0x04 [1] increment - counter C1 := 1 # j++ 0x05 [1] countjump 0x02, 255 # loop back 255 times	[1] load -indirect R0 := [0x100 + R2] [1] setmembndlo 00000011***, 1 [1] load -indirect R1 := [0x300 + R0]

Figure 7: Execution Leases allow indirect memory accesses within bounded memory regions. In comparison, the base GLIFT ISA performs table lookups by iterating over the entire array and predicating out the desired index. For many cryptography algorithms, table lookups are numerous and each base GLIFT lookup adds performance overhead in proportion to the table sizes.

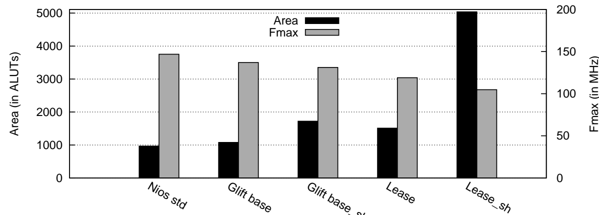

Figure 9: Quantifying the area and timing overhead of Execution Leases in a Glift CPU. The left Y-axis compares the number of FPGA lookup tables(ALUTs) required to implement a Glift Lease processor with that for the Glift base processor and a general purpose RISC micro-processor by Altera. The graph also shows the number of lookup tables to implement versions of both Glift processors augmented with shadow logic for information flow tracking. The right Y-axis compares the Fmax for same set of processors.

largest FPGAs are over 10X larger than EP2S15F67263, and so the area of these CPUs is probably of little concern.

The second result is that in relative terms, the Lease CPU requires 50% more ALUTs than the Glift baseline CPU. Augmenting this Lease CPU with the information flow tracking logic adds an additional 2X ALUTs. This jump from the Lease processor to "Lease_sh", its shadowed version, is much larger than the 70% increase from Glift base to Glift base_sh. Not surprisingly, the main overhead of implementing leases arises from the MUXes and deMUXes required to read and write back to the set of 8 timers, restore PC, restore SP and other lease state. The auto-generated shadow MUXes then account for the overhead of the Lease_sh CPU. Since the lease behavior of a program is known statically, compiler optimizations could potentially bring down the number of hardware stack entries required.

The Fmax of the synthesized CPUs, on the other hand, show that the Glift and Lease CPUs at 120-130 MHz are only slightly slower than the Nios processor at 160 MHz. This difference in frequency is mainly because Glift and Lease processors support barrel shifts while Nios supports only 1-bit shifts. With 1-bit shifts, the Lease CPU can also operate at 160 MHz. In relative terms, the synthesized Lease_sh CPU at 108 MHz also runs slightly slower than the Lease CPU (120 MHz).

5.3 Programming with Execution Leases

To code up our benchmarks, we have designed a simple high level language with constructs that capture the functionality of the Lease CPU. The lexical and grammar definition of Lease compiler is generated using Antlr, and all other parts of the compiler are written in Java. We demonstrate various aspects of our language through an encryption library modeled after the privilege separated OpenSSH. Our library uses an I^2C bus interface for I/O and implements a public-key RSA encryption function to exchange symmetric keys and followed by AES for all subsequent communication. This library models a scenario where the I^2C bus drivers are untrusted (e.g. imported as a commercial binary) and also implemented as a general purpose program with a potentially unbounded loop. The aim is to completely isolate the untrusted drivers and prevent them from ever *affecting* the encryption functions or accessing any information illegally (for e.g., the symmetric key for a transaction).

The `lease` statement is the most important new idea of this language. Its syntax is: **lease**($timer$, $memorysize$, $Function(arg0, ...)$, $returnvalue$); and it allows a programmer to lease a bounded area of memory, jump to a location, and execute a bounded number of instructions. Each function also has an *execution mode*, GLIFT or general purpose, which has to be specified statically.

Estimating Execution Time: The $timer$ argument to a lease statement conveys the number of instructions that the $jumpToFunction$ is allowed to execute. If left at 0 for a GLIFT mode function, the compiler can automatically fill the timer value by statically analyzing the $jumpToFunction$ and computing the total time required to execute both sides of conditional branches, fixed size loops, and some extra instructions to pass in arguments, set appropriate memory bounds, and retrieve the return value. Further research is required to make the compiler capable of estimating execution times for general purpose functions and provide useful suggestions to the programmer of the GLIFT mode caller function. On one hand, many security functions that operate on streaming data are easy to estimate, as they already take a fixed amount of time (e.g. AES, RSA, md5 etc). In cases where the function is accessing some peripheral device (or in general waiting for some asynchronous communication) the lease bounds will be governed by system-level timing constraints. For instance, aircraft require some critical computation every N ms, and the non-critical functions will work around this constraint.

Setting memory bounds: The $memorysize$ is used to bound the local memory accesses $jumpToFunction$ will make, and can either be determined statically for GLIFT mode programs or (as is the case for peripheral interface drivers) be fixed to an arbitrary size based on the system designer's discretion. In addition to local memory bounds, global memory bounds have to be set in case the caller wants to allow the callee restricted access to some additional region of memory. For instance, the I^2C transmit function can be allowed access to the encrypted message in addition to its local frame. The callee can use the `load/store-global` instructions to access these out-of-frame addresses. One concern might be that putting arrays into power-of-2 aligned memory regions might prove prohibitively hard for the programmer. While more research is definitely required to place arrays in the most compact manner possible and to minimize data movement, our compiler has a simple algorithm whereby it determines all the constraint sets for arrays that need to be adjacent, places them along aligned

boundaries as far as possible, and moves arrays around for when all constraints cannot be met. The caller has to specify using the `@` symbol (as in `int [2] @arr`) if a function call argument requires to be accessed using the global load/store instructions.

Another very useful optimization the compiler performs relates to the problem of setting the memory bounds before every load or store, especially in loops. For such cases, the compiler begins by inserting a `setbounds` instruction before every memory access, but in a later pass discovers all adjacent setbound instructions that use the same bounds and coalesces them into a single `setbound` instruction with the common bound.

Execution Modes: Every function declaration requires an @*GLIFT* or @*General* prefix that states the mode the function will execute in. In Glift mode, the Lease ISA only allows a *countjump* instruction to jump to a PC a fixed number of times. The language thus supports fixed size loops of the form `for i in range(start,end,step) { block }`. The usual $if-else$ statements are compiled down to predicated blocks of code, and function definitions and statements resemble similar constructs in other imperative languages. In General-purpose mode, conditional jumps are allowed, and if-else statements need not both execute, but the lease instructions (`settimer` and `setbounds`) are no longer available. General purpose functions can use the conventional in-memory stack to execute function calls, and use general-purpose registers as stack and frame pointers.

In Figure 10, we show a snippet from our encryption library. Execution begins in the main() function in GLIFT mode. This sets a lease for initializing the I^2C bus driver, transmitting the start signal and device address, and reads in input from the serial bus. In the example, we show the serial clock (SCL) and data (SDA) bus lines using the memory mapped addresses (`scl_da[]` and `scl_da_in[]`). Once the input has been read in, either RSA or AES is invoked conditionally, but since the function is in general-purpose mode, it only needs to set a lease of $max(aes, rsa)$ (unlike the Glift-base assembly that would require both AES and RSA to execute on every iteration). It is interesting that even though the I^2C receive function has a loop that queries a device for an ACK and can do so indefinitely, once the lease expires, control will be restored to the main function.

End-to-end property: This example shows how leases can be used by programmers to explicitly manage the flow of all tainted information through the CPU and memory, and thus ensure the *integrity* of some critical function (the encryption library) in the presence of untrusted functions (the bus drivers). Considered in a *secrecy* context, leases could be used to show that the only transfer of information between a secret library and the unclassified bus drivers is the encrypted buffer.

5.4 Quantitative Differences in the Resulting Code

While Execution Leases provide the program with a completely new ability, the option to provably contain the flow of information even when tainted *code* is executing (where that code is tainted either because it is secret or it is untrusted), it also provides quantifiable differences in the performance of applications. We compare the execution time of several different kernels running on the NIOS processor, on the original GLIFT microprocessor, and on the Execution Lease machine. The NIOS code was generated from gcc with level 2 optimizations enabled. The code targeting the original GLIFT machine is hand written assembly. The code targeting the the Execution Lease machine is generated by our custom compiler which performs no optimization other than the lease bound merging discussed above.

As can be seen in Figure 11, the static code size between the different machines are all relatively close (for example MM is 83 instructions instead of the 73 from in the original GLIFT machine) with only a few glaring exceptions (AES is 2X bigger and BSort is 5X bigger). However, even though AES is 2X bigger, it is approximately 68X faster using Leases instead of the original GLIFT ISA. BSort is also interesting because, on the original GLIFT machine, this was really the *only* practical way to do sort – because random indexed lookups took $O(n)$ time instead of $O(1)$. On that machine, bubble sort takes $O(n^2)$ time while merge sort takes on the order of $O(n^3)$ time. On our new architecture, merge sort would take $O(n\log(n))$ time which means that even though bubble sort is larger and takes longer, in fact "fastest worst-case sort" would be a more appropriate benchmark. Measured against the general purpose NIOS CPU, the dynamic instruction counts in a Lease-CPU are comparable, mainly because most of the security kernels are very regular and easy to estimate tightly.

In terms of performance it is worth noting that excluding the two *best* performing applications, the average speedup is still 32%, while the two best performing applications (FSM and AES) each are running 8.1X and 68X faster respectively. FSM and AES are each so much faster because they are dominated by table lookups, FSM to find the next state, and AES to perform the sbox operation. While this shows the potential of Execution Leases to drastically reduce the time to perform some of the most fundamental computations, Leases allow us to compose together larger programs such as the encryption library that would be extremely hard without function calls and extremely slow without table lookups.

6. Conclusions

High assurance systems, while often invisible to their end users, are critical to the safe operation of cars, medical devices, aircraft, military operations, and our modern financial system. The designers of such systems often times wish to be able to prove important properties about their resulting implementation, with non-interference being one of the most desirable and difficult to demonstrate properties. As we have discussed, demonstrating non-interference on a traditional architecture is difficult as all sensitive data, for example the private keys in a public-key encryption system, must be carefully guarded to prevent an adversary from observing or even *inferring* their contents. While preventing such leaks of information is difficult enough in practice, it is even harder to verify formally.

Architectural support for *Execution Leases* has the potential to be a powerful new tool for the designers of high assurance systems. The idea that execution resources are leased out to regions of code with fixed bounds on the time and memory addresses is both a model of execution that a programmers can understand, yet can be implemented in such as way that safety is verifiable all the way down to the gate level. In implementing a full prototype of an Execution Lease machine, we came upon several challenges. For example, execution leases can themselves invoke further leases and the processor has to enforce that successive leases are nested in both time and address space. Bounding all memory accesses within the latest bounds requires constructing the memory address in a unique manner from the bounds and the incoming address, while also requiring

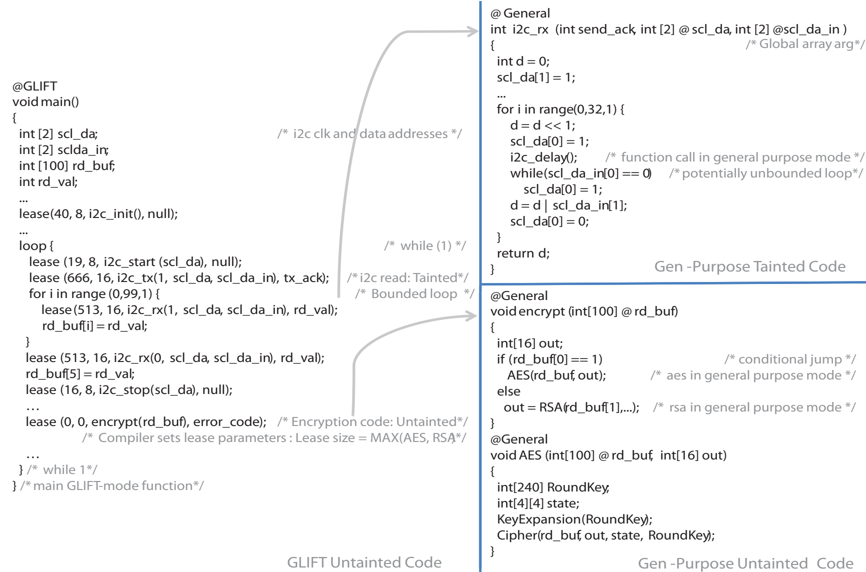

Figure 10: Programming using Execution Leases: Execution begins in trusted (untainted) GLIFT mode. Function calls can be made to general-purpose functions that operate within a fixed time and memory lease. Within a lease, general purpose code can execute conditionals (without predication), potentially unbounded loops, and communicate through protected memory regions, while the CPU implementation guarantees that no tainted code can ever interfere with out of bounds memory or execution time.

a special tag memory that can mark a range of addresses as tainted in parallel. However, we describe a series of techniques for overcoming these challenges, and we show that the resulting machine is both far easier to program and capable of executing far more powerful code than prior GLIFT based processors (it is not limited to GLIFT-ISA from [31]). While there is still more work to do in further refining the ISA, optimizing the hardware implementation, fleshing out the language features, and improving the code generated by the compiler, even now the Execution Lease machine is the only design we are aware of that is capable of demonstrating the non-interference of two or more general purpose software components all the way down to the level of gates.

7. Acknowledgments

The authors would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF Career Grant CCF-0448654, and by grants FA9550-07-1-0532 (AFOSR MURI) and NSF 0627749.

8. References

[1] The integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[2] What does cc eal6+ mean? http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/.

[3] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[4] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *The Cryptographers Track at RSA Conference*, pages 225–242, 2007.

[5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[6] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[7] J. Daemen and V. Rijmen. The design of rijndael: Aes - the advanced encryption standard. 2002.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.

[9] D. Federal Aviation Administration (FAA). Boeing model 787-8 airplane; systems and data networks security–isolation or protection from unauthorized passenger domain systems access. http://cryptome.info/faa010208.htm.

[10] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[11] R. A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, 1983.

[12] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[13] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

[14] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, 2007.

[15] B. Lampson. A note on the confinement problem. *CACM*, 16(10):613–615, 1973.

[16] R. B. Lee, P. C. S. Kwan, J. P. Mcgregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[17] H. M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann, 1984.

Kernel	Description	Static Instruction Count (NIOS)	Static Instructions (original GLIFT)	Static Instructions (Execution Lease)	Dynamic Instructions (NIOS)	Dynamic Instructions (original GLIFT)	Dynamic Instructions (Execution Lease)
FSM	CSMA-CD state machine with with 6 states and 4 inputs. Many table lookups	123	190	130	441	3322	410
BSort	Perform bubble sort on a fixed size list of integers	26	21	126	20621	30358	43518
RSA	Montgomery multiplication and exponentiation from RSA public key cryptography	256	143	95	44880	39297	26329
AES	Block Cipher, involves extensive table lookups and complex control structures	781	1100	2113	12807	1082207	15785
Md5	Core of the cryptographic hash function, involves mostly ALU and logical operations	769	1386	951	1226	1431	1012
MM	Matrix Multiplication	108	73	83	9043*	17035	10094

Figure 11: A comparison between the static instruction counts and dynamic instruction counts between a traditional processor (the Altera NIOS), the original GLIFT based microcontroller, and our Execution Lease based processor. As we have not implemented a multiply instruction yet in our prototype, the Matrix Multiply result for the NIOS does not make use of native multiply instructions either. Excluding the two best *performing applications, the average speedup is 32%, with the two best performing applications (FSM and AES) each running 8.1X and 68X faster respectively.*

[18] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[19] W. Martin, P. White, F. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. *Automated Software Engineering, International Conference on*, 0:133, 2000.

[20] J. Mclean and C. Heitmeyer. High assurance computer systems: A research agenda. In *America in the Age of Information, National Science and Technology Council Committee on Information and Communications Forum*, 1995.

[21] J. K. Millen. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy*, 1999.

[22] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[23] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006*, pages 1–20. Springer-Verlag, 2006.

[24] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[25] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, Dublin, June 2009.

[26] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45, New York, NY, USA, 2008. ACM.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[28] O. Sibert, P. A. Porras, and R. Lindell. An analysis of the intel 80x86 security architecture and implementations. *IEEE Transactions on Software Engineering*, 22(5):283–293, 1996.

[29] G. Suh, C. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *Design and Test of Computers, IEEE*, 24(6):570–580, Nov.-Dec. 2007.

[30] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[31] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[32] TSEC Rainbow Series Library. A guide to understanding covert channel analysis of trusted systems, version 1. Technical Report NCSC-TG-030, Library No. S-240,572, 1993.

[33] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[34] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[35] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[36] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 351. IEEE Computer Society, 2003.

[37] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.