SAFE: A Clean-Slate Architecture for Secure Systems

Silviu Chiricescu[§], André DeHon[*], Delphine Demange[*], Suraj Iyer[§], Aleksey Kliger[§], Greg Morrisett[†], Benjamin C. Pierce[*], Howard Reubenstein[§], Jonathan M. Smith[*], Gregory T. Sullivan[§], Arun Thomas[§], Jesse Tov[†], Christopher M. White[§], David Wittenberg[§],
[*]University of Pennsylvania, Philadelphia, PA [†]Harvard University, Cambridge, MA
[§]BAE Systems, Burlington, MA
Corresponding author: gregory.sullivan@baesystems.com

Abstract—**SAFE is a large-scale, clean-slate co-design project encompassing hardware architecture, programming languages, and operating systems. Funded by DARPA, the goal of SAFE is to create a secure computing system from the ground up. SAFE hardware provides memory safety, dynamic type checking, and native support for dynamic information flow control. The Breeze programming language leverages the security features of the underlying machine, and the "zero kernel" operating system avoids relying on any single privileged component for overall system security. The SAFE project is working towards formally verifying security properties of the runtime software. The SAFE system sets a new high-water mark for system security, allowing secure applications to be built on a solid foundation rather than on the inherently vulnerable conventional platforms available today.**

I. INTRODUCTION. THE CASE FOR CLEAN-SLATE CO-DESIGN

A premise of the SAFE project is that conventional hardware, operating systems, and programming languages are hopelessly broken with respect to security concerns. Attempting to patch an endless stream of newly discovered vulnerabilities, while unavoidable in the short term, is not a viable long-term path to secure computing systems.

The root causes for the current state of affairs can be traced back to decisions made when transistors were scarce, networked computers rare, and formal methods were conducted with pencil and paper. The results of these now outdated assumptions include: computer hardware that can compute impressive rates of floating point operations per second, but has no intrinsic notion of security; programming languages that can describe data structures and functions, but not security properties; and operating systems that are written in unsafe languages and try to enforce security policies on insecure programs running on security-oblivious hardware. The "Trusted Computing Base (TCB)"—that is, the part of a system that, if compromised, would result in a security violation—is essentially the whole system, including hardware, programming language compilers, operating system (device drivers, scheduler, network stack, file systems, etc.), and all application software such as web servers, web browsers, databases, scripting languages, and so forth. The last decade of cyber defense has shown that for every security vulnerability discovered (usually after compromise) and patched, attackers find multiple alternatives in short order. As our cyber defenses grow ever more complex, we seem to be spending many of the computer cycles we have gained over the years towards fruitlessly searching for yesterday's attacks, while continuing to increase the attack surface exposed to attackers.

The SAFE project takes a clean-slate approach to system design. We start by revisiting the historical assumptions listed above, and by applying a "radical co-design" methodology to ensure that security properties are preserved across all layers of the system. The four major thrusts of the SAFE project—hardware, languages, operating system, and formal verification—have been designed and developed in concert to produce a highly secure computing platform, featuring:

1) Hardware support for fine-grained tracking and checking of security properties.
2) Programming languages that support application-level security properties, and compile those properties to the hardware security features.
3) A decentralized operating system consisting of least privilege, mutually suspicious components. The compromise of any single component will not violate system level security guarantees.
4) Formal proofs, using computer-assisted formal methods, that the software stack from applications to hardware maintains end-to-end security properties.

A number of papers have appeared on several aspects of the SAFE project, such as the hardware interlocks [1], multi-level cache algorithms for tag management [2], and error handling in the presence of dynamic information flow control [3]. We presented a paper [4] at the 2011 Workshop on Programming Languages and Operating Systems (PLOS) describing our initial design goals. The present paper represents the results of two additional years of design and implementation; we have an initial implementation of the hardware on an FPGA platform, and we have addressed many of the open questions from the PLOS paper. The reader is invited to visit http://crash-safe.org/, which contains all of the project's publicly

Approved for Public Release; Distribution Unlimited. Cleared for Open Publication on 9/10/2013.

released materials.

The next section walks through two typical vulnerabilities and describes how the SAFE system prevents security violations in the face of such attacks. In the following sections, we present in more detail the security-related elements of the hardware (Section III), the low-level runtime system ("concreteware") (Section IV), and the Breeze programming language (Section V). In Section VI, we present our approach to verification of the concreteware, and in Section VII we give the status of the SAFE project.

II. Motivating Examples

We describe two canonical attacks, (1) buffer overflow followed by binary code injection, and (2) SQL injection, and briefly discuss how the SAFE system prevents these attacks from being successful. While there are obviously a huge range of attack vectors (cf. [5]), we hope that these two examples will provide some intuition for how SAFE addresses vulnerabilities, and will help motivate the technical details presented in later sections.

A. Buffer Overflow and Binary Code Injection

A canonical low-level attack is *buffer overflow followed by binary code injection*. A typical attack sequence consists of:

1) Overflow an allocated region of memory, putting attacker-provided data in overflowed memory.
2) Modify stack to jump (return) to attacker-injected data.
3) Execute attacker-injected data as native instructions, at privilege of the current process.

The SAFE hardware architecture blocks the above attack steps with multiple mechanisms:

1) *Fat Pointers* - Every pointer encodes not only the address to which it points, but also the base and bounds of the frame into which it points. Attempting to index a pointer outside of the frame into which it points results in an error. SAFE uses an efficient fat pointer encoding that extends the work of [6]. The fat pointer mechanism rules out buffer overflow attacks.
2) *User Inaccessible Stack* - User code cannot directly read or write the stack, but rather, can only manipulate the stack through call and return instructions.
3) *Machine-checked Types* - Every word in the SAFE machine has an *atomic group* tag that defines how the word can be used. For example, SAFE has atomic groups for *integer*, *instruction*, *pointer*, and several other specialized types. Thus, even if an attacker could redirect the program counter at words injected via buffer overflow, if the words are not instructions, then the SAFE machine will not execute them. Only the Linker concreteware component has the ability to tag a word as an instruction (more on the "least privilege" runtime design in Section IV), and so data injected by an attacker into a user process will not be executed. The fact that integers cannot be treated as either instructions or pointers, and vice versa, rules out a wide range of attacks to which conventional architectures are vulnerable.

B. SQL Injection (aka "Improper Neutralization")

Buffer overflow and binary code injection attacks can be largely thwarted by hardware mechanisms. However, there are a wide range of attacks that violate higher level security policies and thus need input from the application level to successfully identify and block them. Some well known categories of application level attacks are: privilege escalation, cross-site scripting, and SQL injection.

In a classic SQL injection attack, an input form might request, for example, an employee ID, and then query the database in order to display the corresponding employee name. If the data from the input form is INPUT, the query string might be "SELECT EMP.name FROM Employee WHERE Id = $INPUT". If an attacker enters into the employee ID field something like "1234; SELECT EMP.salary FROM Employee WHERE Id=1234", (a semicolon separates SQL commands) and the application neglects to "neutralize" the input (e.g. throw away anything after a ";"), the attacker may be able to view an employee's salary when they were only supposed to have access to the employee's name.

Both operating system security and programming language security researchers have focused recently on *Information Flow Control (IFC)* as a unifying approach both to *access control* (what authority has access to what data) as well as *label propagation*, namely how to propagate access control metadata as data is combined and flows through a computation. There is a huge range of approaches to information flow – from coarse-grained dynamic per-process labels (*e.g.,* [7], [8]) to fine-grained dynamic per-value labels (*e.g.,* [9], [10]) to statically checked IFC (*e.g.,* [11], [12]).

The SAFE hardware architecture supports the efficient implementation of fine-grained dynamic information flow with two features: (1) tags as pointers, and (2) a programmable *tag management unit*. Together, these features allow the implementation of arbitrary dynamic information flow control policies.

For programs written in the Breeze programming language (Section V), the metadata on values specifies which principals (think of a principal as a user, or an "actor" in the SAFE system) have access to what data. In the SQL injection example, even if a Breeze program neglected to check for input after a semicolon, the information flow control rules would prevent data private to an employee being leaked to an unauthorized channel.

The following section describes the hardware architecture in more detail.

III. SAFE Hardware Architecture

The core idea of SAFE is that security properties of data (and instructions) can be specified at a high level, and faithfully tracked and checked at the hardware level. In order to support this granularity, (1) every word in the system must have metadata associated with it, and (2) information flow properties must be propagated and checked at every instruction.

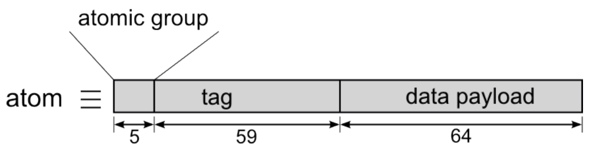

Fig. 1. SAFE atom encoding

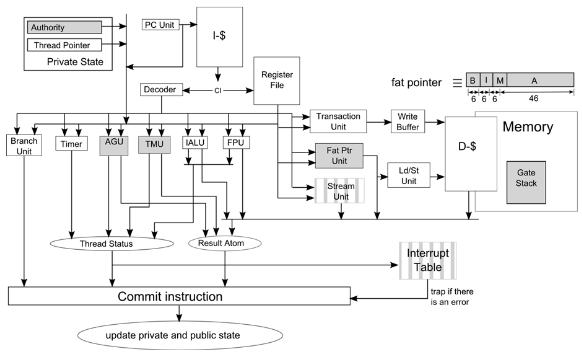

Fig. 2. SAFE architecture

A. Atom = Atomic Group + Tag + Payload

The basic unit of memory on a SAFE machine is called an *atom*. Each atom consist of three inextricably linked parts: the atomic group (described earlier), a *tag* which is a pointer to structured *metadata* (though opaque to user code), and a *payload* which will be treated according to the atomic group element of the atom. Figure 1 depicts the layout of an atom on a 128-bit SAFE machine.

Figure 2 presents a schematic of the SAFE architecture. The architecture specification is implemented in the Bluespec hardware description language [13]. It is mostly a straightforward RISC architecture, with the addition of several security-related components, shaded in gray in the figure, which we now discuss in more detail.

B. Atomic Group Unit

The block labeled "AGU" in Figure 2 is the *atomic group unit*—it is responsible for checking that the atomic groups of the current instruction operands match those required. For example, for an integer ADD instruction, it is required that the atomic group of both source registers be *Integer*.

C. Tag Management Unit (TMU)

The semantics of the SAFE architecture is that at every instruction, the metadata for each atom involved (PC, instruction, and each operand) is evaluated against an installed ruleset and either an access violation is flagged or the metadata for the result is returned. We rely on a TMU rule cache to minimize the overhead of this fine-grained information flow control.

The design used for our TMU cache, to achieve near-associative hash performance, is described in [2]. An overview of the entire suite of hardware interlocks, including fat pointers, atomic group checking, and TMU management, is described in [1].

Critical to our design is that TMU cache processing must happen in parallel to the rest of the machine's execution, and that the TMU cache check must be on the order of a single add instruction so as to not stall the instruction pipeline. This design requirement makes explicit our choice to trade silicon for security, without compromising speed (at least in the common, cache hit, case).

D. Low-Fat Pointers

The box labeled "Fat Ptr Unit" in Figure 2 manages our *low-fat pointers*. The idea of encoding base and bounds in a "fat pointer" has long been a desired feature (*c.f.,* fat pointers in Cyclone [14] or capabilities for memory as in [15]). The SAFE architecture builds on the fat pointer encoding scheme of the Aries project [6], which fits the encoding of a pointer and the base and bounds of the pointed-to frame in a single word. The costs of the Aries fat pointer encoding are (1) some wasted space, because allocation rounds up to 2^n sized blocks (n a parameter), and (2) added encode/decode time in the instruction pipeline. Our initial implementation of the Aries fat pointer encoding scheme resulted in a fat pointer unit that was the largest (in time) component of our pipeline, and so we have invented an enhanced fat pointer scheme, called "low-fat pointers," which breaks fat pointer decoding into two phases and no longer impedes our efforts to pipeline the processor.

E. Lightweight, Fine-Grained Domain Crossing (Gates)

Conventional systems have two (or some fixed, small number of) security regimes—*kernel* and *user*. Kernel mode has complete access to all resources, while user mode restricts access to some system resources such as files and sockets based on authentication. Furthermore, *domain crossing*—switching from kernel to user mode or vice versa—is considered expensive, which encourages pushing more functionality into single system calls rather than separating system functionality into lots of little functions. As kernel mode has "superuser" access to a machine's resources, attackers focus on gaining control while in kernel mode. Some approaches to reducing the vulnerability of so-called "monolithic kernels" include "microkernels", such as the L3-L4 family of microkernels [16], [17], [18]. Microkernels aim to minimize the size of kernel code, under the assumption that a smaller "Trusted Code Base (TCB)" is more amenable to careful validation.

The SAFE system takes a two-pronged approach to reducing the vulnerability exposed by privileged operations:

1) First class authority—We allow for dynamically created *Authorities* that can be used as the basis for isolating privilege.
2) Gate calls—SAFE introduces *gate calls* that atomically switch from one authority to another, with overhead comparable to a simple function call. Lightweight gate calls make domain crossing cheap, which enables a *least privilege* kernel design where each privileged operation in the runtime is segregated to its own gate running under its own authority.

A gate is similar to a *closure* in functional programming language implementations, as it maintains a pointer to a gate's local storage. When a gate call is made, the current authority and local storage pointer are pushed onto the *gate stack*, the authority register is populated with the target gate's authority, and the target gate's local storage pointer is installed. The gate stack is not accessible other than by gate call and gate return operations, and so attack vectors that rely on access to the call stack are impossible.

IV. SAFE CONCRETEWARE. A ZERO KERNEL OPERATING SYSTEM

Above the hardware is the layer of system software referred to as *concreteware*. The concreteware includes thread management and global memory management, providing abstractions of concurrent computation and infinite memory. A key design decision of the SAFE runtime is that there is no user-visible shared memory between threads—all communication between threads takes place via single-reader, single-writer *streams* (a native hardware construct, with an atomic group for *stream pointers*). This distributed runtime shares many elements from the design of Erlang [19].

A. Least Privilege

A central design goal of the SAFE runtime is to ensure that security-sensitive functions run with the *least privilege* required to do their job. We accomplish this goal by leveraging hardware security features including per-instruction access control and gates. For example, there is only one case when a pointer is fabricated (from a larger frame)—in a *memory allocation* gate. The TMU rules, then, enforce the invariant that if a new pointer is being created, the instruction executing that code is running under the authority of the memory allocator. Other examples include: tagging an atom as a *forwarding pointer* can only be performed if running under the *garbage collector* authority; tagging a bit pattern as a *Principal* can only happen in a single function of the PAT server; and extracting the tag component of an atom and treating it as a value can only take place during TMU cache miss handling. As another example, we divide the scheduling process into two subcomponents, each running under its own authority: the *ComputeSchedule* gate calculates the next schedule iteration, based on access to protected data about individual threads, and the *AdvanceSchedule* gate is the single function that can actually swap thread pointers in and out of the machine's state.

The following sections describe the most important components of the SAFE runtime (memory management; management of principals, authorities, and tags; and scheduling).

B. Memory Management

Memory, along with CPU time, is a critical resource to be managed by a runtime system. The SAFE project has made a number of unconventional design decisions around memory management, in the service of the security and verification goals of the SAFE platform.

We have already introduced SAFE's *fat pointers*, which encode base and bounds information within pointers. Furthermore, due to atomic group checking, user code cannot inspect pointers—user code can only dereference pointers. Pointers, then, can be viewed as *capabilities*, as in [15]. That is, the only way that code can access a region of memory is to be handed a pointer to that region of memory; in particular, user code can not fabricate, or "guess", pointers into memory it does not already have a pointer into.

One subtle but important result of encoding pointers as opaque memory capabilities is that there is no longer any reason to use virtual memory as a mechanism for compartmentalizing per-process memory spaces, as is done on most conventional systems. This greatly simplifies reasoning about low level memory management—for example, in the seL4 verification effort [18], one of the places the verification effort had to explicitly compromise their proof standards was around dealing with virtual memory.

There is a *top-level memory manager* that allocates larger frames of memory to individual threads, and which reclaims memory from threads when threads die.

There are many motivations for providing *automatic memory management*, aka *garbage collection (GC)* as a system service in SAFE. For one thing, the decision that threads cannot share memory, which is motivated both by security and formal reasoning concerns, enables a relatively simple implementation of per-thread GC. Similar to the motivation for precluding shared memory, providing automatic memory management supports a layered verification process—we first need to prove that the allocator and garbage collector are correct, and then we can reason about user code without worrying about memory leaks and other memory-related errors.

C. Global Principals, Authorities, and Tags (PATs)

SAFE concreteware supports communication between threads via single-reader, single-writer streams. Because there is no user-visible shared memory between threads, all values are copied (instead of sending pointers across streams). However, it is important that information flow control applies to copies of values the same as to the original values. Recall that *tags* on atoms are pointers to *metadata* about the values. The metadata memory resides in a concreteware thread known as the *PAT Server*, where *PAT* stands for *Principals, Authorities, and Tags*. When values are copied, the tag on the value stays the same – that is, both copies have tags that point to the same metadata, contained within the memory space of the PAT Server thread. In this way, TMU cache miss handling, which applies information flow propagation and access control at every instruction, will perform the same way for the same tags on data in different threads.

SAFE also has atomic groups for *Authorities* and *Principals*. The PAT server thread, in addition to handling TMU misses, also has entry points for providing fresh principals and authorities (one can think of them as unique 64-bit integers, tagged as principal or authority). The terms *principal* and *authority* occur frequently in the information flow control literature,

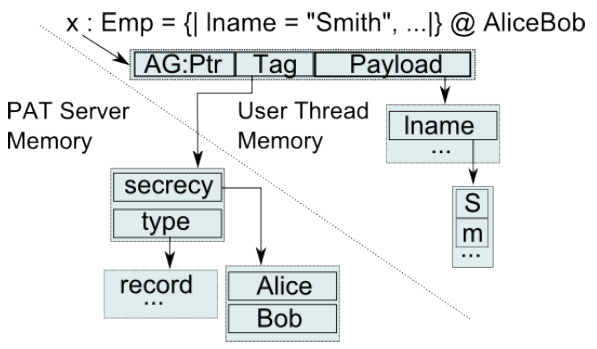

Fig. 3. Schematic of a Breeze value with metadata from multiple label models

and having principals and authorities as first-class values in the hardware enables SAFE to more directly implement information flow control policies. One can view principals as stand-ins for users of the system, or for individual components (actors) of the runtime, and authorities as the "keys" to access values labeled by corresponding principals. The exact structure of metadata, and the correspondence between individual principals and authorities, are functions of a particular *label model*.

In SAFE terminology, *label models* (LMs) maintain the metadata, and multiple label models may be installed at the same time. For example, a value manipulated by a Breeze program may have a tag that points to metadata frames for both the *Breeze Secrecy Label Model* and the *Breeze Dynamic Type Label Model*. The Breeze Secrecy LM maintains nested frames of principals, representing (as formulas in CNF[1]) which principals have read access to the data. The Dynamic Type LM simply contains a pointer to a data structure that provides accessor functions for a data structure's fields. The sketch in Figure 3 represents a value corresponding to a pointer to a Breeze record, whose tag points to both secrecy and type metadata. Note the dotted line indicating that the value and metadata are in two different memory spaces.

The global PAT space also needs to be garbage collected. PAT GC requires a global algorithm that incrementally transitions each thread from an "old" PAT space to a "new" PAT space. The details of this algorithm are beyond the scope of this overview paper.

D. Tempest Programming Language

The concretware is implemented in a combination of SAFE assembly language and a new systems programming language, *Tempest*. Tempest is analogous to the C programming language—sufficiently low level to give direct access to native representations and instructions, but also providing higher level features such as procedure calls, structures, abstract types, and register allocation. Unlike with C on a conventional machine, we get strong type and memory safety guarantees through hardware mechanisms.

[1]CNF = Conjunctive Normal Form. See http://en.wikipedia.org/wiki/Conjunctive_normal_form

Unlike Breeze (Section V), Tempest is typed. In addition to types corresponding to SAFE atomic groups such as integers and pointers, Tempest's type system correctly tracks *linear pointers*. A linear pointer is the only (user-visible) pointer to a frame of memory. On the SAFE architecture, pointers to streams are linear, which greatly simplifies reasoning about concurrency.

Tempest's type system also tracks procedures' calling conventions, which may be specified by programmers. Because several SAFE mechanisms (*e.g.,* the bracket construct in Breeze) depend on careful treatment of registers, different Tempest procedures may pass arguments and results in different registers, and have different registers available for use, thus requiring different calling conventions.

Tempest also provides robust support for inline assembly language, which can refer to Tempest variables in place of physical registers and is properly handled by the register allocator.

V. Breeze Programming Language

Breeze is a new dynamically typed language with first-class labels, authorities, and principals. Breeze's semantics enforce *dynamic, fine-grained information flow control*. That is, every access to a Breeze value checks that the access is allowed given the current authority context and the labels of the data involved. The Breeze language has co-evolved along with the SAFE hardware architecture, in order for Breeze programs to be efficiently implementable on the hardware, as well as to enable programmers to take maximum advantage of the SAFE architecture's security features.

Breeze has a number of interesting features, which we will briefly describe. At the time of this paper preparation, we are working towards releasing Breeze as an open source project – check http://crash-safe.org for more details. The open source release of Breeze will include more detailed language documentation.

A. Public labels and Brackets

Perhaps surprisingly, we have settled on a design in which the labels on Breeze values are public. One might think that a simple exfiltration technique would be to return different labels depending on the value of some secret data. However, in order to test high data but reliably return to a low context, a Breeze program needs to enter a *bracket*, and the label of data returned from a bracket must be specified in advance, thus foiling any attempts to use labels as an information channel.

B. Not-a-Values (NaVs)

Exceptional control flow is difficult to handle in the context of dynamic information flow control. Breeze does not have an exception mechanism, but instead will return a labeled *Not-a-Value (NaV)* value when an access violation occurs. In order to check for an error (*i.e.,* for a NaV), the user code must raise its authority to the label on the value.

Public labels, brackets and NaVs are explored and explained in [10].

C. Clearance

In order to access labeled data, a Breeze program must exercise the appropriate authorities. The mechanism for establishing a context in which secret data can be accessed is *raising the clearance*. The *Labeled IO Monad* in [9] uses a similar clearance mechanism.

VI. Verification of Concreteware

The SAFE project is actively working towards formal verification (using the Coq proof assistant [20]) of the concreteware. The formal property that we are focusing on is *noninterference*. Noninterference formalizes the standard requirement that the high-security inputs to a program should not influence its low-security outputs. For more careful definitions of noninterference, see, for example [21], [22].

As an example of the verification goals of the SAFE project, consider the compilation of a Breeze program. Breeze program values will be tagged with metadata according to the Breeze Secrecy Label model, and the preservation of the semantics of the labels must be maintained by the compiled code and the concreteware services that come into contact with Breeze values.

We are not yet at a stage where we can verify the compiled implementation of the Breeze Secrecy Label Model on the actual SAFE instruction set. Instead, we have defined a relatively simple stack machine and a straightforward compilation of information flow control rules to that machine. We have proved noninterference for the abstract machine, as well as refinement of the abstract machine by the concrete machine plus compiled label model. Even this simplified scenario has presented interesting challenges, and we consider the proof of refinement to have been a significant accomplishment. We are currently working on extending the proof to a larger fragment of the full SAFE architecture.

VII. Implementation Status

As of June 2013, we have implemented the SAFE architecture on an FPGA platform, and we are able to compile simple Breeze and Tempest programs to the FPGA. We currently shuttle TMU misses offboard to a PAT server process running on a linux co-host. In the next few months, we will have the PAT server reimplemented in Tempest and running onboard the SAFE platform. We have a “minimal concreteware” running on a SAFE simulator, but the minimal concreteware does not yet include garbage collection (only allocation). Networking is currently handled by the linux co-host, and we are slowly working towards implementing a network stack on the SAFE processor. To add networking, we also have to implement a more sophisticated scheduler in order to guarantee time to the network device driver (there is only one interrupt in the SAFE architecture—the timer).

VIII. Conclusion

The SAFE project is taking a clean slate approach to designing a secure computing system from the ground up. SAFE provides hardware support for modern security policies based on information flow control, and we trade silicon for security without sacrificing performance. We are working towards formal semantics of the SAFE instruction set architecture (ISA), as well as the Breeze and Tempest programming languages. Furthermore, we are working to formally verify that the SAFE operating system correctly enforces high level security policies such as noninterference.

SAFE will provide a platform that formally guarantees an unheard of level of security from the operating system down to the hardware. While we can never completely rule out programmers writing ill conceived programs, or insider threats, we can at least provide a computing platform, including threading, garbage collection, and high level programming languages, that can guarantee that application level security policies will be correctly enforced.

IX. Acknowledgements

This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, and G. Sullivan, “Hardware support for safety interlocks and introspection,” in *SASO Workshop on Adaptive Host and Network Security*, Sep. 2012. [Online]. Available: http://www.crash-safe.org/sites/default/files/interlocks_ahns2012.pdf

[2] U. Dhawan and A. DeHon, “Area-efficient near-associative memories on FPGAs,” in *International Symposium on Field-Programmable Gate Arrays, (FPGA2013)*, Feb. 2013. [Online]. Available: http://www.crash-safe.org/node/21

[3] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. A. de Amorim, and L. Lampropoulos, “Testing noninterference, quickly,” in *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sep. 2013, to appear. [Online]. Available: http://www.crash-safe.org/node/24

[4] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan, “Preliminary design of the SAFE platform,” in *6th Workshop on Programming Languages and Operating Systems*, ser. PLOS, Oct. 2011. [Online]. Available: http://www.crash-safe.org/sites/default/files/plos11-final_0.pdf

[5] MITRE, “Common weakness enumeration,” http://cwe.mitre.org/, 2013.

[6] J. Brown, J. Grossman, A. Huang, and T. F. Knight, Jr., “A capability representation with embedded address and nearly-exact object bounds,” MIT AI Lab, Tech. Rep. 5, April 2000, aries Project. [Online]. Available: http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-05.pdf

[7] M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, “Information flow control for standard OS abstractions,” in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP. ACM, October 2007, pp. 321–334. [Online]. Available: http://pdos.csail.mit.edu/~max/docs/flume.pdf

[8] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, “Labels and event processes in the Asbestos operating system,” in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP. ACM, 2005, pp. 17–30. [Online]. Available: http://asbestos.cs.ucla.edu/pubs/asbestos-sosp05.pdf

[9] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, “Flexible dynamic information flow control in Haskell,” in *4th Symposium on Haskell*. ACM, 2011, pp. 95–106. [Online]. Available: http://www.scs.stanford.edu/~deian/pubs//stefan:2011:flexible-ext.pdf

[10] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCException are belong to us," in *34th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 3–17. [Online]. Available: http://www.crash-safe.org/node/23

[11] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *Transactions On Software Engineering And Methodology (TOSEM)*, vol. 9, pp. 410–442, October 2000. [Online]. Available: http://doi.acm.org/10.1145/363516.363526

[12] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.7374&rep=rep1&type=pdf

[13] R. S. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE, 2004, pp. 69–70.

[14] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, C. S. Ellis, Ed. USENIX, 2002, pp. 275–288.

[15] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, 1974.

[16] J. Liedtke, "On micro-Kernel Construction," in *15th ACM Symposium on Operating Systems Principles*, 1995, pp. 237–250.

[17] K. Elphinstone, G. Heiser, and J. Liedtke, *L4 Reference Manual: MIPS R4x00, Version 1.11, Kernel Version 79*, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 1999, available from http://www.disy.cse.unsw.edu.au/Softw./L4.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220. [Online]. Available: http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_09.pdf

[19] R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG (2nd ed.)*, J. Armstrong, Ed. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[20] *The Coq Proof Assistant*, 2012, version 8.4. [Online]. Available: http://coq.inria.fr/refman/

[21] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[22] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003. [Online]. Available: http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf