Security guarantees for the execution infrastructure of software applications

Frank Piessens, Dominique Devriese, Jan Tobias Mühlberg and Raoul Strackx
iMinds-DistriNet
KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
Email: {firstname.lastname}@cs.kuleuven.be

***Abstract*—Software applications run on top of infrastructure consisting of hardware (processors, devices, communication networks, ...) and software (operating systems, compilers, virtual machines, language runtimes, databases, ...). In many cases, attacks against application software rely at least to some extent on aspects of that infrastructure, and in some cases vulnerabilities can be fixed by strengthening the infrastructure, as well as by patching the application code.**

This paper argues that it is beneficial for secure software development if the security guarantees offered by the execution infrastructure are explicit and precisely defined. More specifically, a developer writing source code that will be executed on the infrastructure should know what guarantees the infrastructure offers against what class of attackers. We survey existing proposals for precise statements of such security guarantees, and argue that the notion of *full abstraction* proposed by Martín Abadi as a correctness condition for secure implementation of abstractions is the key notion for specifying security guarantees of execution infrastructure.

We give a brief overview of how full abstraction has already been used to specify and prove security for important building blocks of an execution infrastructure, and we sketch a research agenda identifying several interesting open research problems that, when solved, could contribute to a more secure design of execution infrastructure for distributed software applications, and to a better understanding of the security properties of these infrastructures.

I. Introduction

Consider the Heartbleed vulnerability, a buffer over-read vulnerability in the OpenSSL library that could be exploited to stealthily steal the private key of a web server. Who is to blame for such a vulnerability? Clearly the developers of the OpenSSL library carry some of the blame, as the buffer over-read is definitely a bug in their code. But also the C compiler (or language) could be blamed for not limiting the potential consequences of such a bug. A safe language like Java or Rust [1], or a safe compiler for C [2], would ensure fail-safe behavior where reading out of bounds leads to a run-time error rather than to an undetected leak of secrets. And finally also the operating system / hardware could be blamed for not providing more fine-grained memory access control. A capability-enabled processor and operating system [3] for instance could make sure that a pointer to a buffer used for storing network data cannot be used to access a memory segment containing cryptographic keys.

This is just one example of a common phenomenon where attacks against application software rely at least to some extent on aspects of the execution infrastructure. There are many other examples, including for example, all low-level attacks that exploit memory safety vulnerabilities [4], [5], [6], attacks that exploit unprotected network communication (e.g. man-in-the-middle, SSL stripping, tampering with JavaScript loaded over HTTP) and memory scanning attacks that extract secrets, such as credit card numbers or cryptographic keys, out of an application's address space [7].

We loosely define *execution infrastructure* to be the collection of hardware and software that contributes to the execution of a software application represented in source code. So the execution infrastructure encompasses: compilers, API implementations, operating systems, protocol stacks, virtual machine monitors, databases, hardware and so forth.

Attackers can know about, and make use of implementation details of the execution infrastructure. This can amplify the security implications of simple application-level bugs such as the buffer over-read in the Heartbleed example, or introduce application vulnerabilities that are not apparent from application source code, e.g. vulnerabilities to network-level man-in-the-middle attacks.

The claim behind this paper is that it is *essential* for secure development that developers have a clear understanding of the security guarantees offered by the infrastructure. A necessary precondition for such understanding is that these guarantees are explicitly and precisely defined. A design goal should also be that these security guarantees support secure development while being efficiently implementable.

Rethinking and better understanding the security guarantees offered by execution infrastructures is timely, now that more and more software is developed for the heterogeneous collection of networked embedded devices known as the Internet of Things, where there is a much wider variety of hardware and operating systems than for desktop or server applications. This heterogeneity makes it more realistic to experiment with different execution infrastructures that provide different security guarantees.

The remainder of this paper is structured as follows: in Section II we elaborate on the problem statement, and discuss two important existing security notions, identifying *full abstraction* [8] as a key notion for defining the security of

an execution infrastructure. In Section III we briefly survey some existing results on full abstraction, but we also point to several interesting open problems and research questions with respect to the security properties of execution infrastructures for distributed applications. Finally, Section IV concludes the paper.

II. Security guarantees for execution infrastructure

A. Problem statement

A developer working on a software application relies in an essential way on *abstraction*. The programming language is a convenient abstraction of (among others) the hardware processor and memory, and various APIs offer an abstract view of databases, communication networks, and so forth. For the sake of simplicity, let us assume – without loss of generality – that all relevant APIs are modeled as source language primitives. In that case the abstractions that the developer works with are exactly those of the source language, and their meaning can be defined by defining an operational semantics for that source language, for instance in the form of an abstract machine.

These abstractions can be (and often are) relevant for the security of the application being developed. Sometimes this is very explicit, for example when the application uses a secure channel abstraction. But often there are also subtle and implicit dependencies. For instance, abstracting physical memory as a heap of objects is a common abstraction in source languages and security may depend on adequate isolation between two heap objects.

The infrastructure for executing the software implements the source language (including all these abstractions) using techniques such as compilation, interpretation, multiplexing, virtualization, resource-sharing, and so forth.

At run-time, attackers wishing to abuse the software can often interact with the resulting run-time system at multiple layers of abstraction: they may share resources (such as CPU and memory) with the application under attack because they can execute code on the infrastructure too, and they may even have complete control over some of these resources (for instance the network, or removable storage devices).

A key question is: *what security guarantees does (or should) the infrastructure offer with respect to the source language abstractions used by the developer?*

Obviously, the answer to this question depends on the attacker model too. Can the attacker observe or modify network traffic? Can the attacker send input to, or receive output from the application? Can the attacker load code in the same process that the application is running in? We are interested in understanding and improving the security guarantees offered by the execution infrastructure under realistic attacker models.

There are at least two important such guarantees that have been proposed and studied extensively: *language safety* and *full abstraction*.

B. Language safety

The notion of language safety [9] considers the operational semantics of the source code and provides the guarantee that this semantics will never run into undefined behavior. Through a combination of type checks, bounds checks, general precondition checks if APIs are modeled as part of the language, and memory management, the compiler and run-time system together ensure that the execution of any source program is always well-defined. Of course, sometimes, in order to maintain safety, execution will be defined to terminate with an error-condition, for instance when a bounds-check or a precondition check fails.

Language safety is a very useful property. In our initial example, the use of a safe programming language would ensure that upon triggering the Heartbleed vulnerability, the web server terminates with an error instead of leaking the contents of the memory words adjacent to the over-read buffer.

But the attacker model considered by language safety is rather weak. Attackers can only interact with the program in ways defined by the source code semantics, e.g. by providing input to the program or by reading output from the program. This attacker model is appropriate in some scenarios, for instance for a server program running on a physically isolated and well maintained machine that is not concurrently executing software from other stakeholders. Yet, for most application scenarios realistic attackers can do more than just providing input and reading output. In addition, language safety by itself does not provide any confidentiality guarantees.

Essentially, language safety only deals with the case where the attacker *cannot* attack the software system at a lower layer of abstraction than the source code, and in that case it guarantees integrity of the run-time state of the application.

C. Full abstraction

1) What is full abstraction?: A compiler from some source language to a target language is fully abstract if it preserves *abstractions*. Hence a first question is: what exactly is an "abstraction"? Intuitively, an abstraction is a program part (like a function, object or module) that offers some service or functionality to other parts of the program (the *clients* of the abstraction, or the *program contexts* in which the program part is placed). The program part can hide some implementation details of how it offers that service: the same abstraction can typically be implemented in many different ways. For instance, a sorting function is an abstraction that can be implemented based on a variety of different algorithms. A set is an abstraction that can be implemented based on different data structures.

We formally define what an abstraction is in a rather indirect way, by defining when two program parts are the *same* abstraction. We define this as follows: two program parts are the same abstraction if and only if clients (contexts) can not distinguish between the two program parts. For instance, two sorting function implementations implement the same "sorting function" abstraction, if clients can not distinguish them: in any client program using one of the implementations, we

can replace that implementation with the other one without impacting the result of the client program.

Formally, the notion that no program context can distinguish two program parts is captured in the definition of *observational equivalence* (also called *contextual equivalence*). It is sufficient to require that all contexts have the same termination behaviour on the two program parts: if there would be any other observational difference, for instance different return values, we can also create another somewhat bigger program context that will go into an infinite loop or not depending on that return value.

We do not further formalize the notions of "program part" and "program context" as this would depend on the specific programming language under consideration, but we assume that given a program part p and a program context C, we can plug the program part into the context resulting in a full program $C[p]$. Then we can define observational equivalence formally.

Definition 1: Two program parts p_1 and p_2 are observationally equivalent (denoted as $p_1 \approx_{obs} p_2$) if for all contexts C, $C[p_1]$ terminates if and only if $C[p_2]$ terminates.

This definition of observational equivalence in turn implicitly defines what an abstraction is: it is an equivalence class of program parts under the observational equivalence relation. The abstraction of "a sorting function" is the class of all observationally equivalent implementations that perform sorting.

Now a compiler is *fully abstract* if it *preserves* abstractions: if we compile two program parts that are observationally equivalent (i.e. that are the same abstraction), then the compiled program parts should also be observationally equivalent in the target language (i.e. clients programmed in the target language can also not observe any difference between the compiled program parts).

More formally, writing $\lfloor p \rfloor$ for the compilation of program part p to the target language, we require that $p_1 \approx_{obs} p_2$ implies that $\lfloor p_1 \rfloor \approx_{obs} \lfloor p_2 \rfloor$.

In addition, we should require the compiler to be *correct*, which usually implies the inverse of the implication above: if two compiled program parts are observationally equivalent, then the corresponding source program parts must be equivalent. Since this direction of the implication (*reflection* of observational equivalence) is less important for security we do not emphasize it for the rest of this paper. The definition of full abstraction requires both directions of the implication.

Definition 2: A compiler $\lfloor \rfloor$ is fully abstract if it preserves and reflects observational equivalence:

$$\forall p_1, p_2 : \quad p_1 \approx_{obs} p_2 \iff \lfloor p_1 \rfloor \approx_{obs} \lfloor p_2 \rfloor$$

It is important to note that the equivalence $\lfloor p_1 \rfloor \approx_{obs} \lfloor p_2 \rfloor$ is observational equivalence *in the target language*. And the target language may have much stronger observational power than the source language. For instance, if the target language is machine code, then machine code clients can easily distinguish between two different implementations of a sorting function, for instance by reading the memory that contains the compiled code. Many compilers are *not* fully abstract exactly for this reason.

2) Full abstraction for security: Abadi [8] was the first to propose full abstraction as a *security* correctness condition. If an abstraction in the source language hides (or protects the integrity of) some information towards clients in the source language, then a fully abstract compiler will hide (or protect) that information also after compilation against arbitrary clients in the target language. If we think of clients as attackers, that are trying to distinguish two program parts to extract some of the information that an abstraction is hiding, then full abstraction says that any attack that is possible in the target language, is also possible in the source language. In other words, if *no* attacks are possible within the source language, then there are also no attacks after compilation, even if the attacker can interact with the compiled program at the level of the target language.

Keeping this in mind, the intuition behind using full abstraction as a security condition for execution infrastructures is:

- The developer can develop his application in the source language, thinking only about attackers that operate at the level of the source language. He can rely on source language features (such as private local state for objects, or secure communication channels) to build his own application specific abstractions, and it is up to the developer to make sure that clients of these abstractions within the source language can not break intended security properties of these abstractions. An underlying assumption is that attacks that we care about can indeed be modeled as a program context or client that succeeds in distinguishing two implementations of the same abstraction. Agten et al. [10] give examples of how attacks against confidentiality and integrity properties of software can be modeled this way.
- The execution infrastructure is modeled as (1) a target language (at a lower abstraction level) and (2) a translation of source programs to that target language.
  The split between what is modeled in the target language and what is modeled in the translation is determined by the attacker model. We choose the target language *such that target language contexts are a realistic model of what an attacker can do to the execution infrastructure*. One can think of the target language as a model of the execution infrastructure *as seen by attackers*, whereas the source language is a model of the execution infrastructure *as seen by developers*. Roughly speaking, more powerful attackers will be modeled by using a less abstract target language.
- Now, full abstraction provides the following security guarantee: the execution infrastructure ensures that, if the developer avoids all source level attacks against his application specific abstractions, then no attacks are possible against these abstractions when running on the execution infrastructure under the chosen attacker model. In other words no "layer below" attacks [11] are possible

within the chosen attacker model.

Full abstraction is a powerful and versatile correctness condition for the secure implementation of abstractions. It also is a security guarantee that is really targeted towards (and useful for) developers: it supports what Gordon et al. [12] have called the *principle of source-based reasoning*, i.e. if the execution infrastructure is fully abstract with respect to a realistic attacker model, then it is sound to reason about security properties of application software running on that infrastructure based on application source code.

It is worthwhile to emphasize that, while full abstraction is formally a property of a compiler between two programming languages, it is useful to think of the source and target languages not just as programming languages but as *models* of the execution infrastructure. The source language models the execution infrastructure as seen by developers, and hence this model will be closely related to the source language used by the developer, but it may also include models of some of the libraries used by the developer. The target language models the execution infrastructure as seen by attackers. This may include a model of machine code in case the attacker can do code injection attacks, but it will also include models of the security mechanisms used by the compiler as well as very specific primitives that model specific attacker capabilities.

III. State of the art and open problems

Given the interesting properties of full abstraction, it is not surprising that this concept has been studied in various security-related settings:

- Full abstraction was first studied [13] as a correctness condition for cryptographic protocols, where the source language is a programming language that offers some notion of confidential or integrity protected communication channel, and the target language is a programming language with symbolic cryptographic primitives and where communication happens over a public network.
  Target language contexts model attackers that have access to the public network and can handle cryptographic messages symbolically, hence this essentially models a Dolev-Yao style attacker [14].
- More recently, it was also studied by several authors (cf. [15], [16], [10], [17], [18]) as a correctness condition for compilation of programming language abstractions like private local state, objects, closures, and so forth to machine code for a machine that has some memory access control primitives.
  Different authors have studied compilation to address-space layout randomization (ASLR, [15], [16]), to protected module architectures (cf. [10], [17]) and to metadata-tracking processors [19].
- In the setting of compiler correctness and security, full abstraction of typical compiler passes like closure conversion [20], type erasure [21], or CPS transformation [22] have been studied, where the source and target languages are intermediate languages used by the compiler.
- Some security vulnerabilities of the web platform have been studied as violations of full abstraction. Baltopoulos and Gordon [12] describe browser-side tampering with application data as a violation of full abstraction, and propose a solution where the compiler of a multi-tier source language automatically inserts the necessary encryption and authentication for data stored on the browser.
  Fournet et al. [23] show full abstraction of a compiler from an ML-like language to JavaScript, thus making it possible to rely on security properties of the ML code even if that code is executed in a malicious JavaScript context within the browser.

In summary, we understand the security properties of at least some of the building blocks of a typical execution infrastructure.

However, full abstraction is also a challenging property in the sense that it is sometimes hard or impossible to have fully abstract *and* efficient implementations, and that the quantification over program contexts in the definition of full abstraction also makes it hard to *prove* full abstraction in many cases. In the following we discuss some of the research challenges that need to be addressed.

A. Full abstraction can be too strict

Full abstraction is a quite strict correctness condition. This was already noted by Abadi et al. in the first paper that constructed a fully abstract implementation of secure channel abstractions [13]. Since it was not possible in their source language to determine what processes are sending messages to other processes, a fully abstract translation to a target language where attackers can observe the public network needs to build in protections against traffic analysis: target language contexts (attackers) should not be able to learn more than source language contexts. Building in protection against traffic analysis is however expensive in terms of communication cost. It would definitely be interesting to also understand the precise security guarantees that the execution infrastructure offers if it does *not* protect against traffic analysis.

Similar phenomena can be observed in other scenarios where full abstraction is used as the correctness criterion. For example, the secure compiler of Patrignani et al. [17] needs to *mask* object references to make sure a target language context cannot learn anything about the memory layout of a compiled module, and needs to make all modules of the same size to avoid leaking information about the size of modules. These are also examples where achieving full abstraction is expensive, and where it would be interesting to understand the security guarantees one can still have without these expensive countermeasures.

An interesting question is if we can come up with weaker notions than full abstraction that still give a precise and useful security guarantee to developers.

One strategy for addressing this question is to play with the source language: by adding constructs to the source language that expose information (like who is sending messages to who, or how “large” is a given source language module)

full abstraction no longer requires the infrastructure to protect against leaking that information: it now becomes a responsibility for the application developer to implement this protection in source code if it is required.

Recall that the general idea is that the developer is responsible for protecting against attacks that can be modeled as source code contexts, and hence by adding these features to the source code language we shift more security responsibility to the developer and less to the infrastructure.

A very extreme variant of this idea exposes the entire run-time representation of the application state at the source code level, i.e. the source code language is extended with a very strong (but read-only) reflection API. In that case, full abstraction requires the execution infrastructure to only protect the integrity of the run-time state of the application, and not its confidentiality, and often this can be implemented much less expensively.

In ongoing research, we are currently building an execution infrastructure for distributed event-driven applications that is fully abstract in this weak sense for a standard event-driven programming language and for a powerful attacker model. The prototype implementation uses Sancus hardware [24] for isolation and attestation of application modules, and symmetric cryptography for protecting network communication, and the performance and communication overhead is substantially lower than the overhead that would be imposed by an execution infrastructure that is fully abstract in the strong sense.

B. What are the right abstractions at both abstraction layers?

When using full abstraction as a security property of an execution infrastructure, the source language should model the execution infrastructure as seen by developers. Hence it should contain abstractions and primitives that are convenient and simple to program with, such as local private state, or secure channels.

The target language on the other hand should model the execution infrastructure as seen by attackers. Hence it should only contain abstractions and primitives for which it is reasonable to assume that attackers cannot break them. Typical examples include cryptographic primitives and abstractions of security primitives that are implemented in hardware. While the set of cryptographic primitives that execution infrastructure builders can choose from is relatively small and stable, the design space for hardware protection mechanisms is relatively open.

For a given source language, full abstraction can be easy or hard to achieve depending on the choice of primitives in the target language. So a very interesting question is: what kind of hardware security primitives are good for achieving fully abstract implementation of interesting source language features? While we have shown in earlier work, [10] and [17], that it is possible to compile objects fully abstractly to a processor that supports Intel SGX-style memory access control [25], [26], others have recently shown [19] that better fully abstract compilers can be designed that target hardware with tagged memory [27]. We conjecture that hardware that supports capability-style memory protection [3] might even be a better target.

There is an entire design space to be explored here: for each of the source language constructs that one is interested in, it is important to explore the cryptographic and hardware primitives that can implement that construct fully abstractly and efficiently. Parametric polymorphism might require symmetric encryption or a hardware primitive for sealing [28]. Closures and objects might require capability-based memory protection. Linking with strong names (as in the .NET framework) might require primitives for attestation.

Two interesting, open-ended and related research questions about this design space are:

- What are the most versatile hardware primitives for securely and efficiently implementing current programming languages?
- What are the most useful or convenient source languages that can be fully abstractly implemented using the hardware protection mechanisms that are available in processors today?

Alternatively, for execution infrastructures where it may be acceptable to have a larger trusted computing base (for instance the web platform, if one cares mainly about web-level attacks like cross-site scripting, SQL injection or cross-site request forgery), the target language should include models of all parts of the infrastructure that are trusted, including for instance browsers and databases. Then an interesting question is: what security mechanisms should browsers or databases implement to make fully abstract compilation of web applications feasible and practical? This is currently a largely unexplored area.

C. Full abstraction can be hard to prove

The definition of observational equivalence of program parts quantifies over program contexts, and hence the definition of full abstraction quantifies over both source language and target language contexts, and this can make full abstraction hard to prove. Some conjectures about full abstraction have been open for many years. For example the fact that parametric polymorphism can be compiled fully abstractly to an untyped calculus extended with an idealized cryptographic sealing primitive was conjectured more than a decade ago [29], [28], and it is still not proven.

But there are several interesting evolutions in the development of proof techniques for full abstraction.

An important step was the development of fully abstract trace semantics for realistic languages, first developed by Jeffrey and Rathke for the JavaJr language [30]. A fully abstract trace semantics defines the semantics of a program part as a set of interaction traces that are possible with that part, and it has the property that two program parts are observationally equivalent if and only if they have the same set of such possible interaction traces. If both source and target language have a fully abstract trace semantics, proving full abstraction of a translation from source to target can be done by constructing a relation between the set of traces of the

source program part and the set of traces of the corresponding compiled program part.

Another important evolution is that the proof technique of *logical relations* has been developed extensively over the past decade [31]. In [22], [20] and [32], Ahmed and her colleagues have developed proof techniques based on logical relations for proving full abstraction of compilation steps such as closure conversion or CPS translation.

Finally, an interesting novel proof technique was presented very recently [21]: it suffices to construct approximate (in a well-defined technical sense) back-translations from target program parts and contexts to corresponding source program parts and contexts to prove preservation of observational equivalence.

With this arsenal of new proof techniques, one can hope that new and interesting full abstraction proofs are now in reach, and that existing conjectures can finally be proven.

D. Fully abstract APIs specified with program logics

For some of the abstractions offered by execution infrastructures (like objects, types and secure channels), it has been studied how to implement them fully abstractly with existing cryptographic and hardware protection primitives.

But in general the execution infrastructure can offer a wide variety of APIs exposing all kinds of resources (memory, I/O devices, ...) in an abstract way to the developer. Hence, an interesting question is whether there is a general way to guarantee the secure implementation of abstractions offered by such APIs.

A promising approach that can handle a wide variety of APIs is to consider the specification of the API in a resource-aware program logic (like separation logic [33], [34]). These specifications can be seen as the definition of new source level abstractions. The question then is how the execution infrastructure can ensure that this kind of abstraction is secure. At least two promising approaches are being studied. First, the infrastructure can enforce the API contract at run time, using some form of contract checking. Agten et al. have shown how to do this for separation logic [35], but their contract checks are not yet sufficient to achieve full abstraction. Second, the security of the implementation of the abstraction can be verified statically. Jung et al. [36] have recently proposed a program logic that supports proving the correctness of layered abstractions over a shared resource like a network or a heap.

E. Refining the attacker model reflected in the target language

Full abstraction is a useful security property if program contexts in the target language are a realistic model of attackers. Target languages that have been considered by the existing works on full abstraction are only approximations of a realistic attacker model – for instance, they do not model the attacker's capabilities to do certain kinds of side channel attacks, or they model cryptographic primitives in an idealized symbolic manner.

An interesting research question is how to extend these target languages to include more attacker capabilities, while maintaining the feasibility of a fully abstract translation to that target language. Can we model timing side channels adequately by equipping the target language with primitives for measuring time? Can we model attackers that can reboot a system, and show that state-continuity techniques (cf. [37], [38], [39]) can be incorporated in the translation to achieve full abstraction? Can we consider target languages with computational (as opposed to symbolic) models of cryptography?

A harsh reality in security is that once a system is *provably* secure against some attacker model, often real attackers will start coming up with new attacks that fall *outside* of that attacker model. So one can expect multiple iterations to further refine the target language, and the end goal is *not* to come up with a final target level model (i.e. a final attacker model), but instead, the goal is to gain understanding of how the cost of protection goes up as more and more powerful attackers are considered.

This in turn can enable developers to balance the security guarantees they require from the execution infrastructure against the price they are willing to pay for it.

IV. Conclusions and outlook

A rich body of research from the seventies and eighties on operating systems security explores the design space of security guarantees that can be offered by multi-user operating systems, including a variety of access control models (discretionary versus mandatory access control, groups, roles and hierarchies, and so forth). We have built up a relatively deep understanding of how software infrastructure can support multiple users securely.

But we do not yet have the same level of understanding about how software infrastructure can support secure development. What are the exact security guarantees that infrastructure offers to developers and under what attacker models? We have discussed two important examples of such security guarantees: *language safety* which is extremely useful but considers a relatively weak attacker model and addresses integrity but not confidentiality; and *full abstraction*, which is very versatile and provides strong guarantees, but cannot easily be implement efficiently for some attacker models, and is sometimes technically challenging to prove.

This paper argues that substantial additional research efforts from the entire community are required to systematically explore the design space of appropriate security guarantees for execution infrastructures, as well as the design space of cryptographic and hardware protection mechanisms to realize these security guarantees.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven and by the Research Foundation Flanders (FWO). Dominique Devriese and Raoul Strackx hold postdoctoral mandates from the Research Foundation Flanders (FWO).

The authors thank the participants of the Secure Compilation workshop held in August 2016 at INRIA Paris for the interesting and useful discussions closely related to the topic of this paper.

References

[1] N. D. Matsakis and F. S. Klock, II, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: ACM, 2014.

[2] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Everything You Want to Know About Pointer-Based Checking," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015, pp. 190–208.

[3] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[4] U. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security*. Springer, 2010.

[5] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 17:1–17:28, Jun. 2012.

[6] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62.

[7] N. Huq, "Pos ram scraper malware: Past, present, and future," Trend Micro, Tech. Rep., 2015.

[8] M. Abadi, "Protection in programming-language translations," in *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, 1999, pp. 19–34.

[9] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[10] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 171–185.

[11] D. Gollmann, *Computer Security*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[12] I. G. Baltopoulos and A. D. Gordon, "Secure compilation of a multi-tier web language," in *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, 2009, pp. 27–38.

[13] M. Abadi, C. Fournet, and G. Gonthier, "Secure implementation of channel abstractions," in *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, 1998, pp. 105–116.

[14] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[15] M. Abadi and G. D. Plotkin, "On protection by layout randomization," in *CSF*. IEEE Computer Society, 2010, pp. 337–351.

[16] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *CSF*. IEEE Computer Society, 2011, pp. 161–174.

[17] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 6:1–6:50, Apr. 2015.

[18] M. Patrignani, D. Devriese, and F. Piessens, "On modular and fully-abstract compilation," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, 2016, pp. 17–30.

[19] Y. Juglaret, C. Hritcu, A. A. de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Towards a fully abstract compiler using micro-policies: Secure compilation for mutually distrustful components," *CoRR*, vol. abs/1510.00697, 2015.

[20] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, 2008, pp. 157–168.

[21] D. Devriese, M. Patrignani, and F. Piessens, "Fully-abstract compilation by approximate back-translation," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, 2016, pp. 164–177.

[22] A. Ahmed and M. Blume, "An equivalence-preserving CPS translation via multi-language semantics," in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, 2011, pp. 431–444.

[23] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to javascript," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13, 2013, pp. 371–384.

[24] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 479–498.

[25] Intel, *Intel Software Guard Extensions Programming Reference*. Intel, 2014.

[26] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 2–13.

[27] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Pump: A programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '14, 2014, pp. 8:1–8:8.

[28] E. Sumii and B. C. Pierce, "A bisimulation for dynamic sealing," in *Principles of Programming Languages*. ACM, 2004, pp. 161–172.

[29] B. Pierce and E. Sumii, "Relating cryptography and polymorphism," 2000, manuscript. [Online]. Available: http://www.yl.is.s.u-tokyo.ac.jp/ sumii/pub/infohide.ps.gz

[30] A. Jeffrey and J. Rathke, "Java jr: Fully abstract trace semantics for a core java language," in *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, 2005, pp. 423–438.

[31] C.-K. Hur and D. Dreyer, "A kripke logical relation between ml and assembly," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, 2011, pp. 133–146.

[32] W. J. Bowman and A. Ahmed, "Noninterference for free," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, 2015, pp. 101–113.

[33] J. C. Reynolds, "An overview of separation logic," in *Verified Software: Theories, Tools, Experiments*, B. Meyer and J. Woodcock, Eds., 2008.

[34] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 41–55.

[35] P. Agten, B. Jacobs, and F. Piessens, "Sound modular verification of C code executing in an unverified context," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 581–594.

[36] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, 2015.

[37] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[38] R. Strackx, B. Jacobs, and F. Piessens, "Ice: A passive, high-speed, state-continuity scheme," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. ACM, 2014.

[39] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.