Design Space Exploration for Security

Eunsuk Kang
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, USA
eunsuk.kang@berkeley.edu

***Abstract*—Design space exploration involves identifying a set of design decisions, and evaluating their potential impact on various qualities of a system, such functionality, performance, reliability, and security. This activity is widespread in other engineering disciplines, but is rarely articulated or performed during software construction, despite its potential influence on the security of the resulting system. In this paper, we argue that design space exploration should be an essential part of any secure development process. We outline the key elements of a framework intended to support this activity, and discuss the potential benefits and challenges associated with building such a framework.**

I. Introduction

Developers today face a barrage of design decisions when building a software system. These range from architectural-level decisions such as the choice of language frameworks, data allocation, and communication protocols, down to implementation-level decisions such as data structure encodings, libraries, and error handling mechanisms.

A wide variety of design alternatives is both a blessing and a curse, especially in security. We are able to construct increasingly complex and diverse systems at a faster rate than ever; at the same time, understanding the consequences of these decisions, and potential vulnerabilities that they introduce, is well beyond the reach of those but few domain experts.

As an example, consider the following question on Stack Exchange, a popular question-answer site, where a web developer seeks advice on two alternative ways of transmitting a session ID as part of a HTTP request[1]:

> "I recently followed a discussion, where one person was stating that passing the session id as URL parameter is insecure and that cookies should be used instead. The other person said the opposite and argued that Paypal, for example, is passing the session id as URL parameter because of security reasons.
>
> Is passing the session id as URL parameter really insecure? Why are cookies more secure? What possibilities does an attacker have for both options (cookies and URL parameter)?"

What makes this seemingly simple decision challenging for developers? First, security is a type of system property that cannot be readily evaluated or quantified like others, such as functionality and performance. For instance, the above two ways of transmitting a session ID are functionally equivalent, and so a typical test case would not reveal their different impacts on the security of an application. In addition, the sheer complexity of components that constitute modern systems further aggravates the challenge of evaluating a security decision. Foreseeing the potential vulnerabilities of using a cookie versus a URL parameter requires an in-depth knowledge of a modern browser, which takes significant effort to acquire and maintain on a developer's part.

In this paper, we argue that exploring different design decisions and evaluating their impact—an activity called *design space exploration*—should be an essential part of any secure development process. Currently, it is carried out (if at all) in a rather ad-hoc manner, and we believe that opportunities and challenges abound in building frameworks and tools to aid developers in this activity.

We will begin by outlining the general structure of a framework for design space exploration, and the roles that it may play in a secure development process (Section II). We will then report on our initial experience with building and applying a security analysis tool called Poirot (Section III). We will discuss prior works that have influenced our proposal (Section IV), and some of the challenges in integrating design space exploration as a part of a development process (Section V). We will conclude with a discussion and summary of our proposal (Section VI).

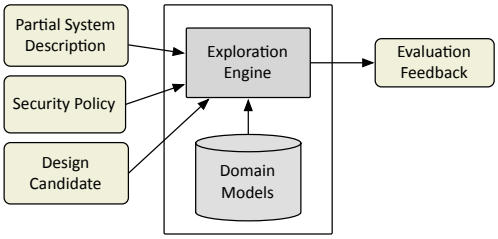
II. A Framework for Design Space Exploration

Conceptually, the idea of design space exploration (DSE) is straightforward: It refers to the act of considering possible options for any decision that contributes to the construction of a system. We, as developers, carry it out all the time (albeit often implicitly in our head). Whenever we debate the trade-offs between using C++ vs Java for our next project, the cost and benefits of deploying SSL for a particular API endpoint, or the allocation of firewall policies, we are all exploring the space of possible design options. Many of these decisions do have significant impact on security, although this is not always evident to those without security expertise or prior experience in the domain. Can we define the notion of *design space exploration* more precisely, and build a systematic framework around to support or partially automate this activity?

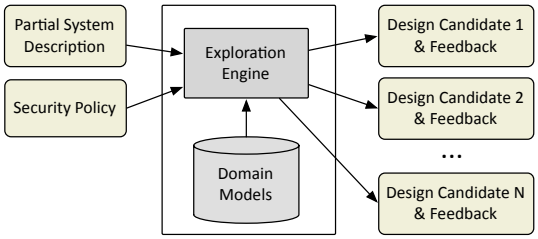
A. Key Elements

A conceptual outline of a DSE framework is shown in Figure 1. The user (typically a developer) provides, at minimum, two different input artifacts: a *description* of the system to

[1] http://security.stackexchange.com/questions/14093/why-is-passing-the-session-id-as-url-parameter-insecure



(a) Usage 1: Evaluating a design candidate



(b) Usage 2: Enumerating design alternatives

Fig. 1. Overview of a DSE framework

be designed, along with a *security policy* that states various restrictions on access to critical system resources. A system description may be *partial*, in that certain parts of the system may be deliberately left unspecified or unknown; these parts are called *design parameters*. A *design candidate* is a particular set of values for design parameters, which, when combined with a partial system description, yields a complete description of the system. Finally, the set of all possible design candidates forms the *design space* for the system.

Recall the Stack Exchange example from Section I. Suppose that the developer who asked the question is attempting to design an online shopping cart application. Informally, the DSE elements in this example would correspond to:

- **System description**: An API model for the shopping cart.
- **Security policy**: "Sensitive data associated with each customer (billing information, cart content, etc.,) should only be accessible to that customer."
- **Design parameters**: The method of transmitting a session ID along with customer requests.
- **Design candidate**: A complete description of how the API is implemented on the HTTP protocol.

We envision two typical use cases for a DSE framework: (1) *evaluation* and (2) *enumeration*. In the former case, shown in Figure 1(a), the user provides a particular design candidate in addition to a system description and a security policy. The *exploration engine* then performs an evaluation of the given candidate, checking whether the system satisfies the given policy; if not, it produces *feedback* that explains how the policy may be violated by a potential attack on the system. For instance, in our example, the user could provide a complete API model that involves transmitting a session ID as a cookie; the DSE engine would then evaluate the design candidate against the policy, and produce a scenario showing how an attacker may carry out a CSRF attack (a possible consequence of using browser cookies) in order to manipulate another customer's account.

Alternatively, the user may not provide any design candidate at all, and simply ask the exploration engine to enumerate potential candidates one-by-one, each time producing feedback that describes potential risks associated with that candidate. (Figure 1(b)). In our example, when prompted without an input candidate, the engine would enumerate all possible ways of transmitting a session ID, and produce an output as follows:

- **Cookie:** Risk of a CSRF attack on stateful operations or cookie theft through XSS.
- **URL query parameter:** Risk of a session fixation attack or exposure of session ID through a HTTP referer header.
- **Hidden form data:** Risk of exposure of session ID through a HTTP referer header if used for GET requests.
- others...

For each of these candidates, the engine would also display a sample scenario that demonstrates a violation of the given policy and, depending on its analysis capability, suggest potential mitigations against the attack. This exploratory mode is especially useful in early development stages, where the developer may have little knowledge or certainty about the consequences of different design decisions.

A key component of a DSE framework is the library of *domain models*, which are leveraged by the exploration engine during the evaluation of potential design candidates. In our example, this library would include generic models of various components of the Web, such as a web server, the HTTP protocol, a browser, and its various features (cookie handling, page rendering, scripts, etc.). Once constructed by domain experts, these models should be *reusable* for analysis of multiple systems in the same domain. The library should also be *extensible*, in that fresh knowledge about a feature or newly discovered vulnerability could be encoded as a separate model and inserted into the library for later use.

So far, we have limited our discussion of DSE to a *conceptual* level, ignoring details such as the underlying notation used to build a system description, the type of security policies allowed, and the technique used for design exploration. In Section III, we will introduce one concrete instantiation of this conceptual framework, and discuss our preliminary experience with DSE.

B. Assumptions

At first glance, what we have described so far does not seem much different from a typical formal verification framework: Given a system model and a security specification from the user, a rigorous technique is applied to check that the system is secure and, if not, produce some evidence (often in form of a counterexample) that shows how the specification may be violated. Indeed, techniques from formal methods will likely play an important role in a DSE framework. However, our proposal takes a departure from traditional verification approaches in several ways:

*a) **Uncertainty in developer knowledge**:* A typical security testing or verification tool is applied *ex post facto*; that is, it assumes that a system has already been constructed, and its complete description (e.g., source code or binary) is available as an input artifact. But in an early development phase, many of the design decisions are yet to be made, and so it may simply be infeasible to come up with any coherent or complete artifact that can be evaluated by existing tools.

A DSE framework must be designed with an assumption that the developer may possess only *partial* knowledge about the system to be constructed. This will require an input notation that allows the developer to express *uncertainty* about design decisions, and an analysis technique that is capable of reasoning about security despite incomplete information about the behavior of the system.

*b) **Design guidance, not proofs**:* The outcome of a typical verification tool is binary: It may produce a positive result, concluding that the given system artifact is free of certain classes of vulnerabilities, or a negative result, highlighting parts of the system that are vulnerable to an attack. Some tools may even provide a proof or a certificate that the system satisfies a certain security policy. Ultimately, the developer expects the tool to produce a *definitive* answer on whether or not the system is secure.

We envision DSE playing a more *informative* role, aiding developers in their decision making process by generating information about potential security risks associated with each design decision. The ability to enumerate design alternatives also enables an analysis of *trade-offs* between them. A design candidate that poses few security risks but overly restrictive in functionality may be passed over for a more flexible alternative with greater but acceptable risks. In some systems, no single candidate may be completely free of vulnerabilities, and so being able to compare alternatives becomes even more crucial.

*c) **Support for evolving decisions**:* A typical development is carried out in a fluid, disorderly manner: Customer requirements change frequently, and the set of available design options may evolve over time. The developer may be required to backtrack on one or more decisions made earlier and explore the reformulated design space. A DSE framework must support this evolutionary nature of design, by showing how a change in a requirement or a design decision impacts security, without the user having to restart the DSE activity from scratch. Most existing verification tools are not designed with this as one of their primary goals.

III. An Experimental System: Poirot

In this section, we describe our initial experience with building a security analysis tool called Poirot [24], which was specifically designed to support DSE activities. This project was motivated by an observation that many vulnerabilities in modern systems are introduced by decisions made during the transition from a high-level design to a low-level implementation. For instance, OAuth [19], a popular authentication protocol formally verified at an abstract level [10], [36], [43], has been shown to be susceptible to attacks when implemented inside a browser or mobile client [40], [12]. The problem is that a standard browser (or a mobile device) contains features and environmental interactions that are absent from the abstract specification of the protocol; and so depending on how the protocol is implemented, different types of vulnerabilities may creep into the final implementation.

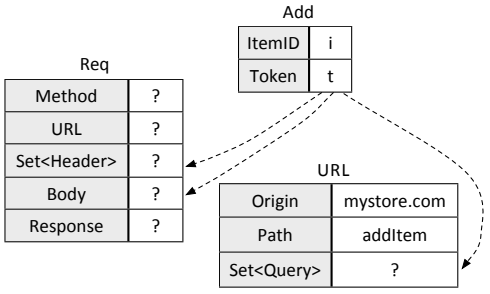


Fig. 2. The structures of the Add operation, a HTTP request, and a URL. The leftmost column in each structure contains the types of arguments to the operation. Dotted edges represent different choices for encoding the token (t) inside a request; "?" represents an unknown value for the argument.

A. Mapping as Design Decisions

A key concept in Poirot is the notion of a *mapping*, which describes how an abstract operation is represented in terms of a concrete operation. For example, when designing an online shopping cart, one may define an operation named Add, which corresponds to the action of adding a new item to a customer's shopping cart. At the abstract design level, this operation contains two arguments, as shown in Figure 2: the identifier of the item to be added (i), and a token that represents a customer's credential (t). In order to deploy the shopping application onto the HTTP protocol, our developer must eventually decide how the two parameters from Add are to be mapped to its counterparts in a concrete HTTP request.

The input language of Poirot provides built-in constructs for specifying these design decisions as a mapping from abstract to concrete operations. More crucially, this mapping may be only *partially specified*, allowing the developer to express her uncertainty about design decisions and systematically explore different candidate mappings. For instance, Figure 2 depicts a partial mapping specification that lists only the origin and path of the Add URL, leaving unspecified how the item ID and token will be transmitted as part of a request; this, naturally, yields a space of possible mappings, each leading to a different implementation of the shopping cart.

Poirot also supports an *incremental* design exploration, by allowing the user to add or remove design decisions without having to modify other parts of the system model. This means, for example, that the user may start with a rather underspecified model of a system, and gradually extend the set of design decisions as she gains more insights about potential security issues through iterative interaction with the tool.

B. Analysis in Poirot

Let us describe the exploration technique behind Poirot in a more precise manner. In the simplest use case, Poirot can

be used as a typical verification tool. Let Abs be a model that describes a high-level system design or protocol, and P a desired security policy. A common method of verifying Abs against P, adapted by Poirot, involves finding a trace of the system that leads to a violation of the policy:

$$\exists t \in T : t \in traces(Abs) \wedge \neg sat(t, P)$$

where $traces(Abs)$ returns the set of all possible traces in the system described by Abs, and $sat(t, P)$ evaluates to true if and only if policy P holds over a particular trace t. If no such trace exists, then we may conclude that the system satisfies the given policy.

Once satisfied with the initial design of the system as depicted in Abs, the user may wish to explore different implementation choices and analyze their impact on security. A key feature of Poirot is in its ability to generate candidate mappings given their partial specification. Let $Conc$ be a model that depicts the behavior of a generic, low-level platform (e.g., a server-browser architecture), and S a *mapping specification* that describes a *partial* relationship between Abs and $Conc$. Then, the problem of generating a design candidate can be formulated as finding a mapping m (adhering to the user-specified S) such that if m is used to implement Abs, the resulting system exhibits at least one trace that leads to a violation of P:

$$\exists m \in M, t \in T : \\ S(m) \wedge t \in traces(impl(Abs, Conc, m)) \wedge \neg sat(t, P)$$

where $impl$ is a composition function that combines two models according to a given mapping. In other words, m describes an *insecure* implementation decision that renders the resulting system vulnerable to an attack.

We are also developing an extension to Poirot to generate a mapping that ensures that the resulting implementation always satisfies the policy:

$$\exists m \in M : S(m) \wedge \\ \forall t \in T : t \in traces(impl(Abs, Conc, m)) \Rightarrow sat(t, P)$$

In other words, the mapping m, if it exists, would correspond to a set of *secure* implementation decisions that together preserve the given policy.

Poirot's engine works by encoding an input system description and policy in a modeling language called Alloy [20], which, in turn, leverages a SAT-based constraint solver to generate sample traces or check a system against a specification. Alloy takes a *declarative* approach, where each model is built as a conjunction of a set of logical constraints, in comparison to an *operational* language where a system is described as a sequence of computational steps. This allows an Alloy model to be *partial*, in that it may depict only parts of a system (and still be amenable to analysis), and constructed in an *incremental* manner, where new details about the system can be simply inserted into the model as additional constraints. These characteristics make Alloy particularly suitable as an underlying formalism for a DSE framework.

We have successfully used Poirot to model and analyze the security of several web applications, discovering previously unknown attacks. We found the ability to reason about security with partial design knowledge especially useful, since we were not the developers of the systems ourselves, and initially started with incomplete knowledge about their behaviors. Building and applying the tool, however, was not without challenges, which we will discussed in Section V. More details about Poirot and its applications can be found in [24], [22].

IV. Related Work

a) ***Verification and static analysis for security****:* State-of-the-art analysis tools can now be applied to realistic systems, and discover domain-specific security issues (e.g., information flow violations) in addition to more common vulnerabilities (e.g., buffer overflows) [31], [45], [46]. Both automated and interactive proof systems have been successfully used to construct *verified* components (e.g., compilers, libraries, operating systems [13], [18], [26], [44]) that provide strong theoretical guarantees about security. As these tools continue to improve in their expressive power and scalability, they will remain an indispensable part of a secure development process.

These tools are typically applied to an existing implementation or system artifact. In comparison, relatively less attention has been paid to building tools that can be used to reason about security during early design stages. Two areas of research stand out: (1) security protocol design and (2) network verification. A long line of work exists on formally verifying cryptographic protocols for desirable security properties [1], [7], [8], [37]. More recently, there has been a growing level of interest in leveraging similar types of techniques for verifying network designs; a number of tools [3], [32], [25] allow the user to analyze the impact of configurations before deploying them onto a real network. Again, the focus of these tools is on *verification*, less on *exploration*, and the notion of a design candidate is often implicit. Nevertheless, theories and algorithms underlying these works are just as applicable to DSE, and we believe that there are clear needs and opportunities for leveraging them for the kinds of exploratory activities that we have discussed in this paper.

b) ***Threat modeling****:* Threat modeling involves identifying and articulating the capabilities of an attacker, and devising countermeasures against potential attacks [29], [39], [41]. Like DSE, this activity is most effective when carried out in an early development phase, where the cost of implementing a mitigation is relatively low.

DSE and threat modeling as complementary activities that are most effective when performed in tandem. The emphasis of threat modeling is on understanding choices available to an attacker in compromising a system (vulnerabilities to exploit, interfaces to use as entry points, etc.). But these choices inevitably depend on the design decisions made by the developer: Different decisions result in distinct system structures and behaviors, which, in turn, give rise to different kinds of threats. Most threat modeling, however, is performed on informal system descriptions, and there seems to be little

tool support for helping developers explore the impact of design decisions on potential threats to the system. The types of analyses that we have proposed as part of a DSE framework may also prove beneficial here by, for example, automatically instantiating generic threat models against a particular system description.

*c) **Security knowledge**:* One of the key lessons from safety engineering that are just as applicable to security is the importance of learning from failures [4]. On the positive side, a wealth of knowledge is available on known vulnerabilities, past incidents, and potential mitigations. Community efforts such as CVE, CWE, and CAPEC have been successfully carried out to collect and categorize different types of domain knowledge into easily accessible databases.

But leveraging these knowledge directly during the construction of a particular system remains a manual, ad-hoc process, in part because most of the knowledge is in a form that is not readily machine-manipulable. There were a few early efforts on integrating vulnerability reports and common security issues into a security tool, such as COPS [15], MulVal [35] and attack graphs [38]. More recently, several researchers have embarked on the task of building a formal, generic model of the Web that can serve as a basis for verification of web-based protocols and systems [2], [5], [16]. In general, however, the problem of codifying and mechanizing security knowledge has received relatively little attention in the research community, despite its importance and applicability in secure development activities, including DSE.

V. Challenges

Much work remains to be done to better understand the needs of developers during early design stages, refine the elements of DSE accordingly, and develop new techniques and tools to support this activity. In this section, we outline a few challenges that we faced during our experience with building and applying Poirot.

*a) **Design Notation**:* One of the major challenges in designing a DSE framework is providing a suitable notation for specifying a design space. Such a notation should be *precise*, so that it can be made amenable to automatic analysis, and ideally also be approachable to developers who may not be able to afford learning an entirely new notation. A programming language may be considered as an option for its familiarity, but is generally a poor medium for articulating design decisions. Formal specification languages, such as Dafny [27] or Alloy [20], provide a more expressive and concise way of specifying designs, but require an advanced knowledge of logic that many developers lack. In fact, this challenge is not unique to DSE; the difficulty involved in building a specification is often regarded as one of the major obstacles to the adaptation of formal methods in industry.

Furthermore, no single notation is likely to be sufficient to capture a wide variety of design decisions that are made during a typical development, and some of those decisions may not even be formalizable. Poirot focuses on one particular kind of design decisions; namely, how to encode high-level operations in terms of low-level representations. But there are a plenty of other decisions with security implications, at varying levels of granularity and abstraction (choice of cryptographic suites, data query sanitization, web server configurations, just to name a few). For example, a detailed analysis of cryptographic suites is likely to require some form of probabilistic reasoning, which lies outside the capability of Poirot.

It is clear that there is no silver bullet that will cover all design tasks. Instead, a more promising approach may be to employ a *domain-specific language* (DSL) that embeds the core design concepts as first-class objects, and facilitates the specification and exploration of design candidates with a minimal amount of syntax. With a growing number of DSLs in areas such as privacy policies [30], [45], web API design [42], [34], and software-defined networks (SDN) [33], [3], we are hopeful that it will be feasible to integrate DSE into a development process without requiring significant effort from developers.

*b) **Improved Feedback**:* As discussed in Section II-B, most useful types of feedback from a DSE tool are likely to involve information beyond a simple yes or no response. Often, the user may wish to examine multiple design candidates at the same time, and evaluate trade-offs that take into account not only potential security risks, but other types of system requirements as well (ease of use, performance, etc.). For instance, a design candidate that has relatively few vulnerabilities but incurs a high performance cost may be considered undesirable, depending on the requirements of the particular system to be developed. A challenge is to provide a feedback mechanism that addresses this multi-dimensional nature of design, by, for example, allowing the user to rank design candidates by certain criteria, and interactively prune out the design space to arrive at a candidate that is deemed adequate for the final implementation.

In addition to information about potential vulnerabilities, a DSE engine could also provide suggestions for mitigating those vulnerabilities. Such a mitigation may be presented in form of additional design constraints, which then can be consulted by the developer during the subsequent implementation step (e.g., "to prevent CSRF, validate each stateful operation using an additional unique session token"). In possible approaches to generating this type of feedback, the DSE engine may leverage domain models that capture different mitigation strategies (e.g., [6]), or apply techniques from automatic model repair ([9], [11], [21]).

*c) **Domain Modeling**:* In our experience with Poirot so far, we have been interested mainly in the security of web-based systems. In order to enable the analysis of such systems, we spent a considerable amount of effort (approximately 6 man-months) building a library of models that describe various components of the Web. The task of building suitable domain models is extremely challenging. First, these models need to be sufficiently general enough to be reusable for multiple systems in that domain. In addition, they need to provide a fairly accurate representation of the reality, since any feedback from a DSE tool will be only as reliable as the fidelity of the

underlying models; for instance, if a domain model omits a crucial system detail that could be exploited for an attack, the tool will not be able to produce any information about the same attack. We believe that a community-wide effort to validate these models and keep them up-to-date will be a worthwhile pursuit that will benefit not only system developers, but other researchers who may be interested in building a similar type of framework.

*d) **Scalability**:* Another challenge faced by DSE frameworks (not limited to software, but in other types of systems as well) is the sheer potential size of a design space that needs to explored. The number of possible design candidates is exponential in the number of design parameters; in some systems, the domain of possible values for one or more design parameters may be unbounded, resulting in an infinite design space. Strategies or heuristics for effectively traversing this space will play an important role in scaling a DSE framework to practical systems.

A number of techniques for dealing with the state explosion problem have been developed by the verification community, and some of those may be applicable here as well [14], [17], [28]. Another type of approach may leverage the developer's insights about the system to guide the exploration engine towards particular regions of the design space. For instance, a DSE framework called Formula [23] allows the user to specify what it means for two design candidates to be considered equivalent, and leverages this information to significantly reduce the number of candidates that it needs to explore. As suggested in Section V-0b, we believe that building a robust feedback loop will be crucial not only for providing useful information to the developer, but also for tackling this scalability challenge.

VI. Conclusion

The term *design space exploration* is a concept that is already well understood in other engineering disciplines. It is routinely carried out in construction of complex hardware, physical, and embedded systems such as aircraft, automobiles, and microprocessors. In these fields, an early investment in design is easy to justify, as manufacturing costs tend to be high, and the consequence of a product failure can be destructive in terms of financial burden (in case of a recall) or potential harm to users. On the other hand, software has been ostensibly considered "cheap" to produce and update, contributing to the rather lukewarm reaction to the idea of upfront design by developers. After all, if a problem can be fixed at any time, why bother spending so much effort trying to prevent it in the first place?

But this attitude may be changing. Computer systems are more connected than ever, with average users storing more of their sensitive data on locations that are easily accessible to malicious individuals with enough will and sophistication. The potential cost of a security failure is rapidly increasing, and no longer limited just to information exposure; with the continual growth in the number of cyber-physical systems and smart devices, a security attack may directly result in damage to critical infrastructures, the environment, and human lives.

Security testing, verification, bug finding, and coding practices will continue to be indispensable tools for ensuring the security of a software system. At the same time, it is of our opinion that security will erroneously remain something of an afterthought—a quality that you add onto an existing system—unless more effort is devoted to articulating DSE as an important development activity on its own, and building tools to aid developers in this activity. Fortunately, our community has made significant progress in developing underlying theories, techniques, and domain knowledge in security; the next step is to put them to appropriate use.

Acknowledgment

We thank our anonymous reviewers for their comments and suggestions, which helped greatly improve this paper.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.*, pages 36–47, 1997.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, pages 290–304, 2010.

[3] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126, 2014.

[4] R. J. Anderson. Why cryptosystems fail. *Commun. ACM*, 37(11):32–40, 1994.

[5] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 247–262, 2012.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 75–88, 2008.

[7] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, 2011.

[8] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96, 2001.

[9] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artif. Intell.*, 112(1-2):57–104, 1999.

[10] S. Chari, C. S. Jutla, and A. Roy. Universally composable security analysis of oauth v2. 0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[11] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros. Abstract model repair. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, pages 341–355, 2012.

[12] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 892–903, 2014.

[13] A. Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 609–622, 2015.

[14] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[15] D. Farmer and E. Spafford. The cops security checker system. *Technical Report, Purdue Department of Computer Science*, 1990.

[16] D. Fett, R. Küsters, and G. Schmitz. An expressive model for the web infrastructure: Definition and application to the browser ID SSO system. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 673–688, 2014.

[17] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181, 2014.

[19] Internet Engineering Task Force. OAuth Authorization Framework. http://tools.ietf.org/html/rfc6749, 2014.

[20] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[21] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 226–238, 2005.

[22] E. Kang. *Multi-Representational Security Modeling and Analysis*. PhD thesis, MIT, 2016.

[23] E. Kang, E. K. Jackson, and W. Schulte. An approach for effective design space exploration. In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems - 16th Monterey Workshop, Redmond, WA, USA, 2010*, pages 33–54, 2010.

[24] E. Kang, A. Milicevic, and D. Jackson. Multi-representational security analysis. In *Symposium on Foundations of Software Engineering (FSE)*, 2016.

[25] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27, 2013.

[26] 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 22nd ACM Symposium on Operating Systems Principles, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220, 2009.

[27] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference (LPAR)*, pages 348–370, 2010.

[28] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[29] Microsoft. Sdl threat modeling. https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx. Accessed: 2016-06-24.

[30] A. Nádas, T. Levendovszky, E. K. Jackson, I. Madari, and J. Sztipanovits. A model-integrated authoring environment for privacy policies. *Sci. Comput. Program.*, 89:105–125, 2014.

[31] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, page 60, 2012.

[32] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, November 7-12, 2010*, 2010.

[33] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 519–531, 2014.

[34] O. A. I. (OAI). Open API specification. https://openapis.org/specification. Accessed: 2016-06-24.

[35] X. Ou, S. Govindavajhala, and A. Appel. Mulval: A logic-based network security analyzer. In *USENIX Security Symposium*, pages 8–8, 2005.

[36] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal verification of oauth 2.0 using alloy framework. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 655–659. IEEE, 2011.

[37] P. Y. A. Ryan and S. A. Schneider. *Modelling and analysis of security protocols*. Addison-Wesley-Longman, 2001.

[38] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284, 2002.

[39] A. Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[40] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. ACM, 2012.

[41] J. Viega and G. McGraw. *Building secure software: How to avoid security problems the right way*. Pearson Education, 2001.

[42] R. Workgroup. RESTful API modeling language (RAML). http://raml.org. Accessed: 2016-06-24.

[43] X. Xu, L. Niu, and B. Meng. Automatic verification of security properties of oauth 2.0 protocol with cryptoverif in computational model. *Information Technology Journal*, 12(12):2273, 2013.

[44] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 99–110, 2010.

[45] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 85–96, 2012.

[46] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, Montana, USA, October 11-14, 2009*, pages 291–304, 2009.