A Security Framework for Executables in a Ubiquitous Computing Environment

D. Llewellyn-Jones, M. Merabti, Q. Shi, B. Askwith
School of Computing and Mathematical Sciences
Liverpool John Moores University
James Parsons Building, Byrom Street, Liverpool, L3 3AF, UK
{D.Llewellyn-Jones, M.Merabti, Q.Shi, R.Askwith}@livjm.ac.uk

Abstract-The vision of ubiquitous computing presents us with unique difficulties in terms of maintaining security. Many of these relate to privacy or quality of service issues, but underpinning these are the requirement for secure execution of code. We present a novel framework for utilising a hybrid method of code analysis combined with component composition techniques. The aim has been to allow sandboxing techniques to maintain security dynamically in a ubiquitous computing environment. We argue that whilst security issues for ubiquitous computing are particularly acute, characteristics of the environment can also be harnessed advantageously to fulfil these security requirements.

I. INTRODUCTION

The essence of ubiquitous computing (ubicomp) is computing that becomes so heavily woven into the fabric of the world and society that it virtually disappears [1]. In order to achieve this a ubiquitous computing environment can be deconstructed into a number of essential but broad characteristics. Banavar and Bernstein [2] highlight three in particular: the Task Dynamism of applications being available everywhere and at all times; the Device Heterogeneity and Resource Constraints of applications moving between devices or devices moving between locations; and the social impact of Computing in a Social Environment. By temporarily ignoring the vision and concentrating first on the technology involved, we are able to provide a better presents us. The Task Dynamism enforces the need for a rich networked and distributed environment in which devices are in constant communication with each other, and a framework allowing applications to migrate seamlessly from one device to another. The Device Heterogeneity and Resource Constraints aspect recognises the variable resources that applications are likely to encounter as they move through a ubiquitous computing environment. The impact of Computing in a Social Environment can be considered in terms of the pervasive nature of the devices and their ability to track individuals and store vast quantities of information in an encompassing manner.

In security terms these all have significant ramifications, amplifying many of the difficulties and concerns to be found in distributed or mobile computing (indeed Saha and Mukherjee [3] see ubiquitous computing as a superset of these two paradigms). We also find similar issues arising in the area of programmable and active networks [4]. The ramifications in terms of privacy are obvious, but by implication the need to maintain security against malicious software or software containing security holes becomes even more pressing.

In addition, general users will not be persuaded of the usefulness of such technology if the benefits are outweighed by the negative security implications of such a system. For current computing environments in which security flaws appear to be commonplace, centralised providers can employ expertise in combating such security issues that a personal user is unlikely to be willing to commit to. Therefore until the likelihood of security breaches can be minimised computing will never be able to become ‘ubiquitous’ in the true and most exciting sense.

Thus, for ubiquitous computing we find a security symbiosis in that the nature of the vision brings with it heightened security requirements, whilst at the same time the vision cannot become reality until these security issues have been properly addressed.

This paper looks at one of the security issues that will need to be addressed if users are to accept the benefits of ubiquitous computing: that of trusting code to be run in a ubiquitous environment, without this compromising the essential security and primary requirements of a Personal Ubiquitous Computing (PUC) user.

Within the current – relatively static – computing environment countless real world examples exist of either malicious executables or legitimate but flawed code causing serious security threats. The former will often take the form of worms and viruses that propagate over the internet, whilst the latter may provide opportunities for hackers or malicious executables to exploit the security holes that result. The effects of a security breach may include quality of service implications in the form of reduced performance, privacy violations, or alterations, corruption or destruction of personal data.

All of these methods have been realised at some point in the past but as yet no truly effective solution has been found to combat them. Although security frameworks for ubiquitous computing exist that deal with issues such as authentication of users [5] and privacy of communication [6], there are few that deal with the issue of preventing the execution of malicious code within the ubicomp context. Those that do are...
using the current static paradigm in which certain rights of execution are granted to certain applications in advance and by expert administrators.

Whilst issues such as privacy and quality of service are of paramount importance to ubiquitous computing, the issue of effective and secure execution of code therefore often precedes them. As such, in this paper we offer a sample framework designed to allow the use of arbitrary code in an effective, fluid and secure manner. In this way the solutions to the tricky privacy issues associated with ubiquitous computing will have a sound base on which to be developed.

This paper begins in section II by briefly discussing some of the currently proposed solutions to the problem. In section III we will argue that in order to provide the best possible security a hybrid solution will be required. In the remaining sections, we develop this into a framework implementation. Some of the difficulties of this system are discussed in section V along with suggested methods for overcoming them and section VI provides a more detailed analysis relating to component composition as used in the implementation. We discuss our current prototype in section VII.

II. CURRENTLY PROPOSED SOLUTIONS

The specific problem being faced in a PUC environment with regard to code execution, is how a particular utility or service that a user wishes to implement within their PUC environment can be trusted not to interfere with the security of the overall system.

This difficulty exists in many computing scenarios and poses problems in areas such as active networks [4], mobile computing [7] and downloadable internet content [8]. However, the problem is particularly acute in a PUC environment for a number of reasons:

- Free flow of data and fluid networks make the security issues potentially more serious;
- With applications moving between devices, from the devices point of view it is necessary to run many executables arriving from potentially unknown sources;
- In a ubiquitous environment a user cannot be expected to understand all of the security issues, and may not even be aware when an application is executed;
- Adoption of PUC technology is dependent on overcoming such problems.

Nonetheless, we also believe that a PUC environment offers a more appropriate environment for tackling some of these difficulties, as we will discuss later. In a general setting there are three main methods proposed for tackling the difficulty of trusted code execution. These are sandboxing, certification and Proof Carrying Code.

A good overview is provided by Rubin and Geer [9], but in order to provide some context we will briefly describe each of these methods here.

A. Sandboxing

Sandboxing involves executing code in a restricted and safe environment, particularly by restricting access from the code to operating system functionality [10]. A crucial feature of this technique must be to ensure that there is no way for the executing code to overcome or bypass the limitations imposed by the sandbox. This can be achieved in a number of ways, for example by preventing arbitrary code execution by enforcing type safety, or by allowing greater control of the execution environment by running the code on a virtual machine. In terms of safety, sandboxing can be highly effective, however this comes at the expense of flexibility [11, 12] as the process is heavy handed in its approach, with restrictions being set to the maximum required in order to enforce security in all circumstances. This means that potentially dangerous functions (such as saving to disk or utilising network connections) are restricted to all programs even if many would actually use such functions safely.

B. Certification

Certification generally involves trusting the opinion of someone else that a particular piece of code is safe [13]. The certification process is able to guarantee the source of the program, or the source of some opinion expressed about the program by making use of asymmetric encryption signatures. However, the burden of whether to trust the code or opinion originator remains with the code executor. Most often the system operates with code being signed by a large corporation – that responsible for the production or distribution of the software – in order to guarantee the origin of the executable. Trust is then maintained by the appreciation that a large corporation’s reputation may be reliant on secure code and hence it is assumed reasonable to trust the executable on this basis.

C. Proof Carrying Code

Proof carrying code (PCC) strikes an effective balance between security and flexibility. The process, pioneered by Necula and Lee [14], involves the code producer attaching additional data to a piece of code. This data can be interpreted as a proof that a particular property holds for the piece of code. A recipient of the code is able to verify the proof and check whether or not it is valid and, if it is, they therefore are guaranteed (that is: they know with certainty) that the property holds for the piece of code. The technique requires no certification, since the validity of the proof can be established independently from the trustworthiness of its source. It also exploits the serendipitous property that verifying a proof can be achieved quickly and efficiently. There are nonetheless a number of downsides associated with proof carrying code. In particular, it requires the code producer to go to the effort of adding a proof to the piece of code chosen for execution. Moreover, because the proofs are pre-determined by the code producer and are dependent on the particular property being proven, the potential user of an
executable is only able to verify the properties for which there are proofs.

D. Direct Code Analysis

Perhaps the ideal solution, were it to be made practical, would be to undertake direct analyses of the executable code on the client machine in order to establish the properties of a piece of code. This obviates the need to adhere to an externally determined security policy (as would be the case with PCC), and it also removes the need to resort to external trust of organisations, as with certification. At the same time it can embrace the full flexibility and security aspects of PCC. The problem with such a solution is mainly that of practicality: verifying PCC proofs may be fast, but establishing a proof in the first place is resource hungry and may be time consuming. We will discuss Direct Code Analysis in more detail in section V.

III. HYBRID SOLUTION

Each of the techniques already described has merits and drawbacks. We believe that in order to satisfy the security demands of a PUC environment a hybrid solution will be required, drawing on elements from each of the techniques and extending them in an integrated way with the addition of new methods. We also argue that it is the most effective way to take advantage of the PUC environment, comprising multiple varied networked devices.

We can see that no single solution is satisfactory by considering their individual drawbacks. On the one hand we have the sandbox technique that is self-contained, but is too restrictive. On the other hand the remaining techniques allow for more flexibility, however these also require additional input provided from the code originator, be it certification or added data for proof construction. This additional data may not be available in all circumstances. A situation therefore arises whereby only specially compliant code can be utilised without it affecting the usefulness of the application.

Yet an ideal situation would even allow code that had no additional elements added to be tested and potentially run without restrictions, if it is indeed safe.

Our framework has therefore been designed to present such a hybrid solution, making use of the techniques discussed in the last section augmented by using the ubiquitous and networked nature of a PUC environment.

The general outline of this framework running on a single device is shown in Fig. 1. The diagram depicts the data flow through the framework and, in particular, the data processing as it is applied to the extended executable (which we view as a piece of data). An extended executable is a piece of executable code that may have been supplemented with additional security information. We will describe the format for such code in detail in the next section, but in general this may include certificated properties, proof information or other data.

![Fig. 1. PUC security framework](image-url)
PCC, component composition and certification. It also makes use of novel techniques: Direct Code Analysis, and secure component composition. We will now look at the design and methods used in more detail.

IV. IMPLEMENTATION

In order to understand how security properties are established within our framework implementation, we must first consider what we refer to as an extended executable structure.

This allows executable code to be enhanced using stipulated security properties, PCC data and signatures to provide additional but optional information that the framework we are describing will utilise.

A crucial factor that has been considered as part of the design is that the process should never require the additional data provided as part of an extended executable, it should only ever be used as a beneficial but unnecessary extension to the “vanilla” code. It will be possible to assess the successful fulfilment of this design goal once we have described our implementation.

An extended executable has additional segments of security properties, property proofs and signatures appended to the code, along with an additional code header allowing the extended properties to be properly delineated from the main body of code. An example extended executable is shown in Fig. 2a. The additional properties are optional and can be added to (but not amended due to the signatures) later by linearly appending additional data to the end of the structure. Thus, a more complex structure following the same synopsis for an extended executable is as follows:

\[ \text{extended executable} = \text{code} \mid \text{header}, \text{block} \]

\[ \text{block} = \text{code} \mid \{\text{code}, \text{A-properties}, \text{A-proof}\}_{KA} \mid \text{block}, \text{X-properties}, \text{X-proof} \]

\[ \text{code} = \text{a description of the “vanilla” executable code} \]

\[ \text{header} = \text{a description of the structure of the data} \]

\[ \text{X-properties} = \text{a description of the properties established of the code by actor X} \]

\[ \text{X-proof} = \text{a PCC style property proof of the properties established by actor X} \]

The intention of the extended executable is to reduce the resource requirements needed in order to establish the properties of the code, particularly for devices such as PDAs with minimal resources. In such circumstances the signed source would allow a PDA to establish code properties without the need to undertake any additional analysis.

In the event that no such extensions have been added to the code, however, the situation remains that they must be established in some other way. To see how this can be achieved, we must consider the data processing stage entitled ‘Add Signed Property Extensions’ carefully.

As can be seen from the diagram shown in Fig. 3, the code – possibly already with property extensions – enters the data processing section and its signed properties are checked against those required for the security framework. If suitable extensions exist and have been signed by an actor trusted by the framework, the code is immediately output for execution; no further checking is required. However, it is possible that the data will be plain “vanilla” code, or will contain extensions that are unsigned, or signed by a distrusted source.

In this case further processing is required. If the data contains a PCC style property proof then we first attempt to verify the proof to establish properties in this way. Proof verification requires minimal resources [14] and so we attempt the verification natively within the framework on the device. If the proof verification is successful, the data is extended by appending property extensions containing details of the verified properties. These properties are also signed by the framework and the code is then output for execution. Since the extended executable now contains property details signed by a trusted source (the executing device), the executable may now enter the second stage.

However, it may be the case that the proof fails to verify, or that no property proof is contained in the data. In this case, further processing is required as is again indicated on the diagram. In this situation, the intention is to establish the properties using direct code analysis, but it may not be possible to do this on the device being used, due to resource limitations. It is under such circumstances that the networked...
nature of the framework becomes beneficial. Crucially, if the code does not already include extensions signed by a trusted source, it is able to send the code via the network to a trusted device (generally another device within the cluster of a personal network) that does have the resources to undertake an analysis of the code. In its simplest form, this process might be as follows: The code is passed on to a trusted external device with instructions to establish properties for the code. In order to establish the properties, the external device utilises the same process from the framework used during the code execution as shown in Fig. 3. Once this external device has analysed the code and established its properties it then returns the data to the original device having appended the properties as a signed extension. This extended executable then goes through the whole of stage 1 again on the original device, except that now – having been signed by a trusted device – the code is immediately output for execution.

Because the external device uses the same framework process as described, then we see that the process may be repeated: resource restrictions may cause the code to be sent on to a third device. This process may continue recursively, so that ultimately a chain of trust is established by the extended executable signature additions. Thus from the code executor’s point of view the properties have ultimately been signed by a trusted device, even though this trusted device did not itself undertake the analysis.

Stage 1 is designed so that the extended executable leaving the process will always contain property extensions signed by a trusted source. Before considering stage 2, we will first discuss the process of Direct Code Analysis in greater detail.

V. DIRECT CODE ANALYSIS

In general direct code analysis is not considered to be a viable solution to the problem of establishing the properties of an arbitrary piece of executable code. However, practical advances that have been seen mainly in the field of PCC suggest that direct analysis may be practical under certain circumstances. There are two stages to the establishment of properties under the PCC paradigm. The first involves producing verification conditions for a particular piece of executable code, and the second involves producing a proof that the verification conditions are indeed true. For PCC the verification conditions must be established on both the user’s computer and the code producer’s computer, whilst proof construction need only be undertaken by the code producer (once discovered it is sufficient for the user’s device to simply verify the proof; a much more straightforward task). However, if both processes can be utilised on the user’s computer applied just to the executable code of a program, then we would have achieved a system of Direct Code Analysis. Two difficulties materialise when this is attempted, one for each stage. In terms of establishing a proof, the biggest difficulty is that of time and resource requirements.

However a great deal of research has gone into automatic proof generation, and systems are now at the stage where this may no longer be prohibitive. Experiments undertaken by Colby et al. [15] give an indication of the current time penalties for undertaking code analysis. Using a certifying compiler that adds verification conditions and proofs to code during compilation, their results show that these additional (non-conventional) elements took between 19% and 53% of the overall compile time. These results were obtained on real world example programs. Nonetheless, such a process would be beyond the scope of many portable devices in a PUC environment, hence utilising distribution as described in the framework set out earlier becomes essential.

In terms of verification condition generation, there is a particular difficulty in establishing such conditions from executable code involving backwards branching. This is because, in order to produce the verification conditions the executable code is converted into propositional logic using a method described by Floyd [16]. Using this method, in order to generate the Hoare conditions for looping-code, in general the loop invariant must be known. The method chosen to overcome this in the PCC context is to have these loop invariants added as comments to the compiled code, either as part of the compilation process or as a task for the programmer. Having done this the loop invariants can easily be discovered during the construction of the verification conditions.

However, in the general case when verification conditions must be established by arbitrary code for Direct Code Analysis, this is not an option.

There are certain cases where the loop invariants can be established, or the problem circumvented. For example, we have an implementation involving direct loop unravelling that will work in certain circumstances. Using such techniques a practical solution may therefore be attainable. With the inclusion of loop invariants, good results have been obtained by Necular, Lee, Colby and others [14, 15, 17, 18].
VI. COMPONENT COMPOSITION

During section III we briefly talked about the process involved at stage 2 of the implementation as shown in Fig. 1. This stage is primarily concerned with component composition and the use of this to ultimately dictate the sandboxing methods used during the execution of an application. We gave an example situation involving potential buffer overruns. Compared to many component composition properties this example was overly simplistic, and the system currently being developed will in general be able to utilise much finer grain and intricate component composition techniques, such as those described by Shi and Zhang [19]. To achieve this we use a simple XML application to associate component composition patterns with sandboxing techniques.

For the example cited from Shi and Zhang [19] the technique must match with a strict component hierarchy. An XML template for such a pattern is shown in Fig. 4. The format is both simple and powerful, allowing a variety of graphs to be specifically matched. This includes distinguishing cycles and restrictions on the number and properties of ancestor and predecessors of a node in a component composition pattern. Simplicity is maintained by allowing tree and finite or repeating topologies to be easily defined. Flexibility is maintained by providing a Turing complete language, ensuring any set of computable topologies can be described. Fig. 4 includes comments to explain the format the component patterns must conform to.

Although the system described here provides a basis on which to apply more concrete component composition techniques, it does not itself provide any details as to which compositions apply to which sandbox restrictions. This is a potentially wide area of research and many applicable results exist. For example we have already mentioned Shi and Zhang [19]; others include McLean [20], Focardi and Rossi [21], Mantel [22], and many more. In general these results take a common approach that states that a particular property holds for a composed application given that the individual components also satisfy the property individually. This situation amounts to one in which the properties of a composed application are the properties of the weakest elements.

We take the view of Shi and Zhang [23] that an effective approach to component composition requires greater flexibility in the configurations that are considered. Specifically, they show that it is not necessary for the weakest components to dictate the properties of a composed application. It is this flexibility that we are attempting to address by proposing a generalised method for linking configurations to sandbox methods. For further discussion of these issues, see Askwith et al [24, 25].

VII. PROTOTYPE

Our current prototype has concentrated on stage 2 of the process. We have successfully used component composition analysis to distinguish Composable Assurance [19] based on an XML template similar to that shown in fig. 4. This can be demonstrated for any composed application, allowing for dynamic sandbox configuration. We also have a Direct Code Analysis method implemented, but this is still at an initial stage of development and has yet to be integrated into the system and with other analysis techniques.

At this stage, whilst this process can be successfully applied on a single device, we have yet to employ distributed analysis of components. Nonetheless, successful component composition analysis provides a basis for us to develop these additional elements as part of a complete prototype.

Fig. 4. XML example pattern describing a strict hierarchy
In this paper we have looked at the need for security in a PUC environment and some of the currently proposed techniques designed to protect devices from code with security flaws or malicious intent from causing damage. We have argued that in a PUC environment a hybrid approach is appropriate both in order to obtain the benefits of each of the methods and also to make best use of the PUC environment that it would operate in. We proposed a novel implementation of such a system incorporating these techniques and discussed how Direct Code Analysis and Component Composition techniques could be implemented as part of the framework. We finished by discussing some of the difficulties with Direct Code Analysis, provided novel solutions for some of these difficulties and argued that they would not be insurmountable given appropriate further research.

REFERENCES